Symbolic execution and the testing of COBOL programs

Thesis

SYMBOLIC EXECUTION AND THE TESTING OF COBOL PROGRAMS

A thesis submitted to the

Department of Computing at the Open University

for the degree of

Doctor of Philosophy

by

Philip David Coward

B.Sc.

October 1991

Date of submission : 1st November 1991

Date of award : 1st February 1993

ABSTRACT

The thesis is in two parts. Part one is a review of existing work in the area of software testing and more specifically symbolic execution. Part two is a description of the symbolic execution testing system for COBOL (SYM-BOL). Much of the work presented has been published or accepted for publication.

Part one commences by introducing the aims of software testing and is followed by a review of the tools and techniques of software testing that have been developed over the past 25 years. A simple taxonomy of software testing techniques is given. One potentially powerful technique is symbolic execution. The principles of symbolic execution are described followed by the problems in applying symbolic execution. Part one is completed by a review of existing symbolic execution testing systems. No symbolic execution testing system has previously been built for a commercial data processing language such as COBOL. Part two commences by outlining the features of the SYM-BOL system and describes the user strategies that may be employed when using the system.

The system generates an intermediate form in stages by transforming the source program into one that contains only a limited number of language constructs. Path selection can be automatic or undertaken by the user. In both cases the results of the symbolic execution already undertaken are available to the path selector to help reduce the likelihood of selecting an infeasible path. A description of how the Nag-library linear optimizer E04MBF is used for feasibility checking is given. Feasible solutions are turned into files of test cases. Simple assertions may be included in the source program which do not affect the normal execution of the software but which can be verified by inclusion in the symbolic execution.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Thanks are due to my two supervisors Professor Darrel Ince of the Open University and Professor Mike Hennell of Liverpool University. Their advice, reading of drafts and encouragement has been invaluable. I urge other supervisors to arrange the occasional lunch in an idyllic country restaurant on warm July days. Such supervisory meetings are a major morale booster to the flagging part-time PhD student.

I also thank Ray Norman for his invaluable help with the latest COBOL standard and Alan Price for drawing the flowgraphs.

Thanks are also due to Doctor Necdet Sanerkin for his meticulous care and attention when checking punctuation and grammar. Any remaining errors are of course the responsibility of the author and typesetter.

I would also like to thank the examiners Professor Keith Bennett of Durham University and Professor Dick Housden of the Open University for their thorough preparation and helpful suggestions.

Many other people have helped in a variety of ways, I am indebted to them all.

PUBLISHED PAPERS AND SEMINAR PRESENTATIONS

1988 Coward, P.D., 'Determining path feasibility for commercial programs', SIGPLAN notices, 1988, 23(3), pp93-101.

1988 Coward, P.D., 'A review of software testing', Information and Software Technology, 1988, 30(3), pp189-198.

1988 Coward, P.D., 'Symbolic execution systems - a review', Software Engineering Journal, 1988, 3(6), pp229-239.

1990 Coward, P.D., 'Software testing techniques' in The Software Life Cycle, Butterworths, 1990, pp386-402.

1990 Coward, P.D., 'Symbolic execution and testing' New directions in Software Development 1990 - Testing large software systems, Wolverhampton Polytechnic, March 7th 1990.

1990 Coward, P.D., 'Symbolic execution and testing', Colloquium on software testing for safety critical systems, IEE Savoy Place London, June 19th, 1990.

1991 Coward, P.D., 'Path feasibility, linear optimizers and the evaluate standard form', SIGPLAN notices, 1991, 26(1), pp47-56.

1991 Coward, P.D., 'Symbolic execution and testing', Information and Software Technology, 1991, 33(1), pp53-64.

1991 Coward, P.D., 'Symbolic execution and software validation', Spring Seminar Series, Bournemouth Polytechnic, 15 May 1991.

1992 Coward, P.D. and Ince, D.C., 'The role of symbolic execution in software maintenance', Journal of Software Maintenance, 1992, 3(4), pp183-192.

RESEARCH AIMS

At the outset the principal aim was to undertake a broad literature survey to review software testing techniques with a view to establishing an area in need of further work. One such area was symbolic execution which did not appear to have been considered in the area of commercial data processing software. The research aims from this point were as follows:

1. Undertake a literature survey to review the existing symbolic execution testing systems and establish their strengths and weaknesses.
2. Identify the problems facing the application of symbolic execution to commercial data processing software and in particular to COBOL.
3. Propose means of overcoming the problems in creating a COBOL symbolic execution testing system.
4. Devise an approach to path selection that:
    a. selects more useful paths than existing symbolic execution systems;
    b. utilises the results of symbolic execution in a bid to reduce the likelihood of selecting infeasible paths.
5. Identify problems facing the use of a linear programming routine to assess the feasibility of paths and to overcome these problems demonstrating the practicality of the technique in a COBOL system.

6. Demonstrate that these proposals are practicable by constructing a prototype symbolic execution testing system for COBOL. It is not considered possible within the time-scale of a Ph.D. to build a full system.
7. Evaluate the prototype symbolic execution testing system for COBOL.
8. Identify further work necessary to turn the prototype into a full working system and to identify areas in need of further research.

PART ONE

REVIEW

CHAPTER ONE INTRODUCTION

This thesis is concerned with program testing and in particular with symbolic execution as a technique for aiding program testing.

The term 'testing' is often used to describe techniques of checking software by executing it with data. A wider meaning will be attributed to testing in this thesis. Testing includes any technique of checking software, such as program proving as well as the execution of test cases. Checking implies that a comparison is undertaken. The comparison is often made between the output from a test and an expected output produced by the tester. The expected output is usually based on the specifications. The tester derives the expected result by hand.

Two terms often associated with testing are 'verification' and 'validation'. Verification refers to ensuring correctness from phase to phase of the software development cycle. Validation involves checking the software against the requirements. These strategies have been termed the horizontal and vertical checks. Sometimes, verification is associated with formal proofs of correctness, whilst validation is concerned with executing the software with test data. This thesis avoids these terms and instead refers only to testing.

Program testing is one of many activities that go to make up the larger complex task of software development. The development of a software product is a series of transformations from one level of abstraction to the next culminating in executable software. The need for program testing arises out of an inability to guarantee that the earlier transformations have been performed adequately. Testing is an activity which attempts to assess how well earlier

transformations have been performed.

Software development commences when the commissioner identifies a need. Producing software which provides the required functions will not necessarily satisfy all the requirements placed upon it. Additional requirements may be that the system satisfies legal obligations or that it performs within specified response times or that it meets documentation standards or that the software is written to a particular house style to enable easier modification. This thesis addresses the testing of functional requirements rather than non-functional testing.

Functional testing may be employed when testing a new program or when testing a program that has been modified. 'Regression testing' is the name given to part of the functional testing that follows modification. Primarily, regression testing is undertaken to determine whether the correction has altered the functions of the software that were intended to remain unchanged. There is a need for the automatic handling of regression testing. Fischer describes a technique and associated software tool for determining which tests need to be re-run following a modification [Fisc77].

It is easy to define levels of abstraction through which software development passes. One such series is as follows:

* requirement definition;
* requirement specification;
* software design;
* software.

A program which contains faults will not be congruent with the software design. The software design may in turn contain faults preventing it matching the requirements specification, and so on. Even the first level of abstraction, the requirement definition, contains faults in that it is not congruent with the 'real' requirement.

To help overcome this tendency for increasing fault incidence with the levels of abstraction, testing should take place at each level. Discovery of faults at a particular level of abstraction may cause development to return to a previous level or several earlier levels of abstraction before the source of the problem is discovered.

Testing at each level of abstraction is not easy. The software can be tested by executing it with test cases. This is not generally possible for the higher levels of abstraction where alternative techniques are necessary. There are some exceptions to this where specifications written in notations such as VDM can be executed, albeit inefficiently, and hence tested in much the same way as the software itself [Hekm85]. Advances in formal specification have brought greater confidence in the correctness of specifications and designs but most software developers still use execution of the software as the final demonstration of correctness.

1.1 Rapid Prototyping

One approach to functional testing that has received attention recently is that of prototyping. A prototype is a system or program that works in the sense that it is capable of accepting input data and processing it to produce output. This does not imply that the system is fully

'working'. A prototype is not expected fully to satisfy the requirement or meet non-functional requirements such as levels of performance.

Prototypes differ from working systems in that working systems are in use [Dear83]. A prototype, on the other hand, may perform only a limited range of the functions required. It is a development tool. There may be more than one prototype, each explores some part of the required functions.

The purpose of a prototype is to allow the rapid creation of a system that performs in a way similar to that required of the working system. The prototype is then used as a communication tool between the commissioner and the system developer. When the commissioner has difficulty specifying the requirements a prototype provides a tangible 'object' on which to base discussions. The problems encountered with the prototype allow the creation of the working system with the problems removed.

Rapid prototyping can be viewed as a technique for assessing the higher levels of abstraction from the requirement definition down to the software design. It is not a tool for testing the software immediately prior to launching a 'working' system. The deliverable that arises from creation of a prototype is a specification that is more reliable than would otherwise be the case. This thesis does not consider the testing of prototypes.

1.2 Software product testing

A software product consists of several components: the executable software; software documentation; user documentation; etc. All of these components are to be fit for purpose before release of the software product and so each requires testing. This thesis is concerned with testing only the software component.

The literature is not united about the aims of software testing. Testing encompasses detecting errors during development and checking the requirements. The goals of testing by execution are unclear. On the one hand, testing is concerned with finding faults in the software, on the other it is concerned with demonstrating that there are no faults in the software, though it is difficult to see how this may be done other than by formal verification.

These differing perspectives may be viewed as an individual's attitude towards testing which may have an impact on how testing is conducted. Aiming to find faults is a destructive process, whereas, aiming to demonstrate that there are no faults is constructive. Adopting the latter strategy may cause the tester to be gentle with the software thus giving rise to the risk of missing inherent faults. The destructive stance is perhaps more likely to uncover faults because it is more probing. Weinberg [Wein73] suggests that programmers regard the software they produce as an extension of their ego. To be destructive in testing is therefore difficult. NASA appears to have believed this for many years having, in 1975, established teams of software validators separated from the software creators [Spec84].

An alternative view is that testing passes through two distinct phases. Few people with any experience of writing programs would begin testing by expecting the program to be correct, i.e. without any faults. The first phase of testing is where errors are expected to be detected. Once these errors have been found the cause, i.e. the faults in the software, must be found. The location of faults - debugging - is an associated but distinct activity. On locating a fault, it must be corrected and testing continued.

Gradually, as the detection of errors becomes less frequent, the role of testing changes. A second phase of testing commences in which the aim is to demonstrate that the program is now correct, i.e. free from faults.

Precisely when the transition from phase one testing to phase two testing takes place is not easy to define. When the tester begins to feel confident that the faults have been removed seems intuitively correct, but we do not have a measure of this confidence. Similarly, it is difficult to state the circumstances when the test reverts to phase one when a fault is discovered in phase two.

There are many questions concerning testing which are difficult to answer. How much testing should be undertaken? When should we have confidence in the software? When a fault is discovered, should we be pleased that it has been found, or dismayed that it existed? Does the discovery of a fault lead us to suspect that there are likely to be more faults and, the more faults we find, the more we suspect are left waiting to be discovered. At what stage can we

feel confident that all, or realistically most, of the faults have been discovered? To what extent is testing concerned with Quality Assurance? What is the relationship between testing and the creation of fault tolerant software? In short, what is it that we are doing when we test software?

Testing is about both finding faults AND demonstrating their absence. The aim is to demonstrate the absence of faults. This is achieved by setting out to find them. These views are reconciled by establishing the notion of the 'thoroughness of testing'. Where testing has been thorough, faults found and corrected, re-tested with equal thoroughness, then we have established confidence in the software. If, on the other hand, we have no feel for the thoroughness of the test we have no means of establishing confidence in the results of the testing. Much work has been done to establish test metrics to assess the thoroughness of a set of tests and to develop techniques that facilitate thorough testing. These are discussed in Chapter 2.

CHAPTER TWO TOOLS AND TECHNIQUES OF TESTING

There are many widely differing testing techniques. But, for all the apparent diversity they cluster or separate according to their underlying principles. There are two prominent strategy dimensions: functional/structural and static/dynamic. A purely functional strategy uses only the requirements defined in the specification as the basis for testing, whereas a structural strategy is based on the detailed design or implemented source code. A dynamic approach executes the software and assesses the performance, whilst a static approach analyses the software without recourse to its execution.

2.1 Functional versus structural testing

A testing strategy may commence either from the specification or from the software. When starting from the specification the required functions are identified. The software is then tested to assess whether the functions are provided by the software. This is known as functional testing, not to be confused with the functional v non-functional categories of testing. Alternatively, commencing from the software, the structure is identified and used to derive test cases which in turn are used to assess whether the software meets the specification. This is known as structural testing. When developing a new program based on an existing program where only a subset of the functions are required it is all to easy to accidentally incorporate unnecessary functions. These functions which are included in the software, but not required, are more likely to be identified by adopting a structural testing strategy in preference to a functional testing strategy. The converse may be true for errors of omission.

2.1.1 Functional testing

Functional testing involves two main steps. First, identify the functions which the software is expected to perform. Second, create test data which will check whether these functions are performed by the software. No consideration is given to HOW the program performs these functions. This approach is used during system and acceptance testing.

There have been significant moves towards more systematic creation of requirement definitions, specifications and design [DeMa81, Haye87, Jack75, Jone86]. These may be expected to lead to a more systematic approach to functional testing. Rules can be constructed for the direct identification of function and data from systematic design documentation. These rules do not take account of likely fault classes. Weyuker and Ostrand [Weyu80] suggest that the next step in the development of functional testing is a method of formal design documentation which includes a description of faults associated with each part of the design as well as the design features themselves.

Howden [Howd81] suggests this method be taken further. It is not sufficient to identify classes of faults for parts of the design. Isolation of particular properties of each function should take place. Each property will have certain fault classes associated with it. There are many classifications of faults. A detailed classification is given by Chan [Chan79] and is a refinement of Van Tassel's [VanT78] classification. Chan's classification consists of 13 groups which are subdivided to produce a total of 47 categories. Kaner [Kane88] also identifies 13 major categories which are subdivided giving a total of over 400 specific errors.

Functional testing has been termed a black box approach as it treats the program as a box with its contents hidden from view. Testers submit test cases to the program based on their understanding of the intended function of the program.

An oracle is someone who can state precisely what the outcome of a program will be for a particular case. Such an oracle does not always exist and at best only imprecise expectations are available [Weyu82]. Simulation software provides a powerful illustration of this problem. No precise expectation can be determined and the most precise expectation that can be provided is a range of plausible values.

2.1.2 Structural testing

The opposite strategy to the black box approach is the white box approach. Here testing is based upon the detailed design or source code rather than on the functions required of the program, hence the title structural testing. This approach is used during unit testing and integration testing.

Whilst functional testing necessitates the execution of the program there are two possibilities for structural testing. First, and most commonly practised, is to execute the program with test cases. Second, and less common, the functions of the program are compared with the required functions for congruence. The second of these approaches is characterized to some extent by symbolic execution and more precisely by program proving.

Structural testing that involves execution of the program may require the execution of a single path through the program or that a particular level of use has been made of all the code. The notion of a minimally-thorough test has occupied research efforts over the years. The level of what constitutes a minimally-thorough test of a program has progressively been increased as follows:

* All statements in the programs should be executed at least once [Mill63].
* All branches in the program should be executed at least once [Mill63].
* All LCSAJ's in the program should be executed at least once [Wood80]. An LCSAJ (linear code sequence and jump) is a sequence of code ending with a transfer of control out of the linear code sequence.

Miller [Mill84] has listed a range of 13 structure based coverage measures ranging from

* execute all statements in a program;

through

* execute all subtrees in the hierarchical decomposition tree for the program;

to

* domain testing of a path using the paths input variables.

Achieving each of these is necessary for a good test to be performed on a program. To achieve a given level of coverage requires that all earlier level metrics have been achieved. The best test is an exhaustive test where all paths through the program are domain tested. Here domain tested means that N+1 tests are executed on the path boundary and two tests off the boundary for each predicate on the path where N is the number of input variables on the

path. There are two obstacles to this goal which account for the existence of the other measures listed above.

The first obstacle is the large number of possible paths. The number of paths is determined by the numbers of conditions and loops in the program. All combinations of the conditions must be considered and cause a rapidly increasing number of combinations as the number of conditions increases. Loops add to the combinatorial explosion and give rise to an excessively large number of paths. This is most acute when the number of iterations is not fixed but determined by input variables.

The second obstacle is the number of infeasible paths. An infeasible path is one which cannot be executed due to the contradiction of some of the predicates at conditional statements. It is surprising that, in a sample of programs, of the 1000 shortest paths only 18 were feasible [Hedl85].

```
1   accept A
2   if A > 15
3   then
4     compute B = B + 1
5   else
6     compute C = C + 1
7   end-if
8   if A < 10
9   then
10    compute D = D + 1
11  end-if
```

Figure 2.1 A program fragment

Consider the program fragment in figure 2.1. There are four paths through this fragment as follows:

Path 1 lines 1,2,3,4,7,8,11.

Path 2 1,2,5,6,7,8,9,10,11.

Path 3 1,2,5,6,7,8,11.

Path 4 1,2,3,4,7,8,9,10,11.

Path 1 can be executed so long as the value of A is greater than 15 after the execution of line 1.

Path 2 can be executed so long as the value of A is less than 10 after the execution of line 1.

Path 3 can be executed so long as the value of A lies in the range 10 to 15 inclusive after the execution of line 1.

Path 4 cannot be executed regardless of the value of A because A cannot be both greater than 15 and less than 10 simultaneously. Hence this path is infeasible.

Even trivial programs contain a large number of paths. Where a program contains a loop which may be executed a variable number of times the number of paths increases dramatically. A path exists for each of the following circumstances:

* where the loop is not executed;
* where the loop is executed once;
* where the loop is executed twice... etc.

The number of paths is dependent on the value of the variable controlling the loop. This poses a problem for a structural testing strategy. How many of the variable-controlled-loop-derived

paths should be covered? Miller and Paige [Mill74] sought to tackle this problem by introducing the notion of a level-i path.

A level-0 path leads from an entrance to the software to an output without employing any branch more than once. Any loop that exists on the level-0 path is executed only once when it is executed. A level-1 path is a series of consecutive branches that have already been included on a level-0 path. Thus a level-1 path is part of a level-0 path that is to be repeated just once. Similarly, a level-2 path is part of a level-1 path that is to be executed once more than on the level-1 path, ie. three times.

A possible structural testing strategy might attempt to execute the greatest level path, say level-greatest path. By attempting to execute level-greatest path first, many of the branches on lesser level paths (ancestral paths) will be collaterally executed. In other words, when executing the deepest level-i path many other ancestral paths will necessarily be executed as part of that execution, thus reducing the number of tests required to execute the remaining unexecuted branches.

Because the testing of every path is generally impossible, branch coverage is commonly used as a more practical metric. However, achieving a high branch coverage is not a simple matter, the main hindrance being infeasible paths. The difficulty becomes apparent when a feasible path is sought for a particular branch. Many of the selected paths may be found to be infeasible and pinpointing a feasible path can require much careful searching.

A further difficulty in achieving complete coverage of a testing metric is the presence of island code. This is a series of lines of code following a transfer of control or program termination and is not the destination of a transfer of control from elsewhere in the program. An example of island code is a procedure that is not invoked. Island code should not exist. It is caused by an error in the invocation of a required procedure, or the failure to recognize redundant code following a maintenance change.

2.2 Static versus dynamic analysis

A testing technique that does not involve the execution of the software with data is known as static analysis. This includes program proving, symbolic execution and anomaly analysis. Program proving involves rigorously specifying constraints on the input and output data sets for a software component such as a procedure. The proof is created by demonstrating that each sequence of steps in the procedure causes the input to be transformed to the output. Symbolic execution creates expressions for the output variables on a path in terms of input variables and constants. Anomaly analysis searches the program source for anomalous features such as island code.

Dynamic analysis requires that the software be executed. The goal is to achieve a certain level of program test effectiveness. For example, a coverage metric such as coverage of branches in a program may be used to assess the effectiveness of a test. Test data is created and, following execution, the output is compared with the expectation. Following a test execution the values for the program test effectiveness metrics are reported. Dynamic analysis could be

described as a form of automatic documentation of software execution.

The recording of test effectiveness metrics during dynamic analysis relies on the use of probes inserted into the program [Knut73, Paig74]. These probe statements make calls to analysis routines which record the frequency of execution. As a result the extent of statement, branch, LCSAJ and any other coverage metric can be reported on completion of execution. The code not exercised by the testing is listed.

Assertions about the values of variables can be incorporated at particular points in the program. Should these assertions be violated during execution the dynamic analysis would report the details of the violation.

Dynamic analysis can act as a bridge between functional and structural testing. Initially functional testing may dictate the set of test cases. The execution of these test cases may be monitored by dynamic analysis. The program can then be examined structurally to determine test cases which will exercise the code left idle by the previous test. This dual approach results in the program being tested for the functions required and, the whole of the program being exercised. The latter feature ensures that the program does not perform any function that is not required.

2.3 Taxonomy of testing techniques

Over the last 15 years many testing techniques have been established. There is no generally accepted testing technique taxonomy. The degree to which the techniques employ a static v dynamic analysis or a functional v structural strategy varies and provides the basis for a simple classification of testing techniques. The grid in figure 2.2 outlines one classification. The techniques are described later in the chapter.

2.3.1 Static-structural

No execution of the software is undertaken. Assessment is made of the soundness of the software by criteria other than its run-time behaviour. The features assessed vary with the technique. For example, anomaly analysis checks for peculiar features such as the existence of island code. On the other hand, program proving aims to demonstrate congruence between the specification and the software.

2.3.1.1 Symbolic execution

Symbolic execution, sometimes referred to as symbolic evaluation, does not execute a program in the traditional sense. The traditional notion of execution requires that a selection of paths through the program is exercised by a set of cases. In symbolic execution, cases consisting of actual data values are replaced by symbolic values. A program executed using inputs consisting of actual data values results in the output of a series of actual values. Symbolic execution on the other hand produces a set of expressions, one expression per output variable. Symbolic execution occupies a middle ground of testing between executing

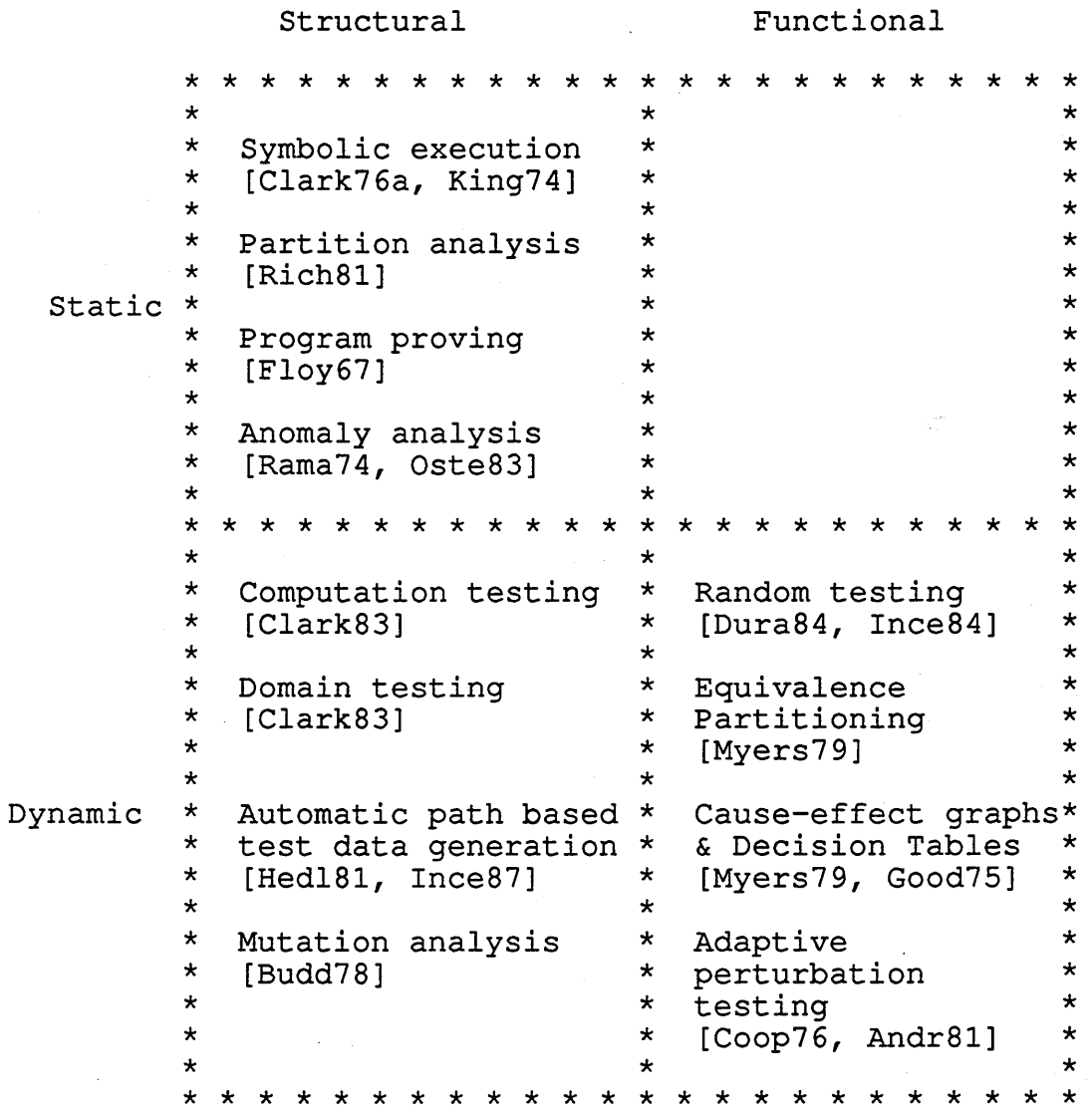

Figure 2.2 Classification of testing techniques

with test data and program proving.

There are a number of symbolic execution systems, for example see [Boye75, Clar76a, King76, Rama76]. The most common approach to symbolic execution is to perform an analysis of the program resulting in the creation of a flowgraph. This is a directed graph which contains decision points and the assignments associated with each branch. By traversing the flowgraph from an entry point along a particular path a list of assignment statements and branch predicates is produced. The execution part of the approach takes place by following the path from top to bottom. During this path traverse, each input variable is given a symbol in place of an actual value. Thereafter, each assignment statement is evaluated so that it is expressed in terms of symbolic values of input variables and constants.

Consider path 1, 2, 3, 4, 5, 6, 8, 9, 11, 12 through the program fragment in figure 2.3. The symbolic values of the variables and the path condition at each branch are given in the right-hand columns for the evaluation of this path.

At the end of the symbolic execution of a path the output variables will be represented by expressions in terms of symbolic values of input variables and constants. The output expressions will be subject to constraints. A list of these constraints is provided by the set of symbolic representations of each condition predicate along the path. Analysis of these constraints may indicate that the path is not executable due to a contradiction. This infeasibility problem is encountered by all forms of path testing.

```
                                        Path Condition    A     B     C     D
 1  accept A                            -                 a     -     -     -
 2  accept B                            -                 a     b     -     -
 3  accept C                            -                 a     b     c     -
 4  accept D                            -                 a     b     c     d
 5  compute A = A + B                   -                 a+b   b     c     d
 6  if A > C                            a+b<=c            a+b   b     c     d
 7  then compute D = D + 1
 8  end-if                              a+b<=c            a+b   b     c     d
 9  if B = D                            a+b<=c            a+b   b     c     d
10  then display 'Success' A D
11  else display 'Fail' A D             a+b<=c AND b<>d   a+b   b     c     d
12  end-if                              a+b<=c AND b<>d   a+b   b     c     d
```

Figure 2.3 Program fragment and symbolic values for a path.

Ideally, a series of assertions for output variables should be produced prior to detailed design. The output expressions from symbolic execution are examined to ensure that they do not conflict with the assertions. In the example of figure 2.3 only the symbolic expressions for A and D will be of interest as these are the only output variables.

A major difficulty for symbolic execution is the handling of loops (or iterations). Should the loops be symbolically evaluated once, twice, a hundred times or not at all? Some symbolic executors take a pragmatic approach. For each loop three paths are constructed, each path containing one of the following:

* no execution of the loop;
* a single execution of the loop;
* two executions of the loop.

Symbolic execution does not specify the number of paths that should be considered, nor is there a set of criteria for selecting paths to execute symbolically, but coverage metrics may be used to assess thoroughness.

2.3.1.2 Partition analysis

Partition analysis uses symbolic execution to identify sub-domains of the input data domain. Symbolic execution is performed on both the software and the specification. The path conditions are used to produce the sub-domains such that each sub-domain is treated identically by both the program and the specification. Where a part of the input domain cannot be allocated to such a sub-domain then either a structural or functional (program or specification) fault has been discovered. In the system described by Richards [Rich81] the specification is expressed in a manner close to program code. This is impractical. Specifications need to be written at a higher level of abstraction if this technique is to prove useful.

2.3.1.3 Program proving

The most widely reported approach to program proving is the 'inductive assertion verification' method after Floyd [Floy67]. In this method assertions are placed at the beginning and end of selected procedures. "A procedure is said to be correct (with respect to its input and output assertions) if the truth of its input assertion upon procedure entry ensures the truth of its output assertion upon procedure exit." [Hant76].

There are many similarities between program proving and symbolic execution. Neither technique executes with actual data and both examine the source code. Program proving aims to be more rigorous in its approach. The main distinction between program proving and symbolic execution is in the area of loop handling. Program proving adopts a theoretical

approach in contrast to symbolic execution. An attempt is made to produce a proof that accounts for all possible iterations of the loop. Some symbolic execution systems make the assumption that if the loop is correct when not executed, when executed just once and when executed twice then it will be correct for any number of iterations.

Analysis of the source code is performed in the same manner as for symbolic execution, but the goal is not just a single expression for an output variable. Instead, many points along a path are selected and assertions concerning the state of variables at these points are made. The source code analysis produces expressions at the selected points and a comparison is made between the derived expressions and the assertions. One of the selected points is usually chosen to be a point of output for the path, and at this point the goal of program proving is the same as for symbolic execution, namely, the output of expressions representing variables.

Program proving can be summarized by the following steps:

* Construct a program.
* Examine the program and insert assertions at the beginning and end of all procedure blocks (definition of blocks is arbitrary).
* Determine whether the code between each pair of start and end assertions will always achieve the end assertion given the start assertion.
* If the code achieves the end assertions then the block has been proved correct.

> If the code fails to achieve the end assertion then mistakes have been made in either the program or the proof. The proof and the program should be checked to determine which of these possibilities has occurred and appropriate corrections made.

DeMillo et al. [DeMi79] describe how theorems and proofs can never be conceived as 'correct' but rather, only 'acceptable' to a given community. This acceptability is achieved by their being examined by a wide audience who can find no fault in the proof. Confidence in the proof increases as the number of researchers, finding no faults, increases [Laka76]. This approach has clear parallels with the confidence placed in software. The wider the audience that has used the software and found no fault the more confidence is invested in the software.

When a program has been proved correct, in the sense that it has been demonstrated that the end assertions will be achieved given the initial assertions, then the program has achieved partial correctness. To achieve total correctness it must also be shown that the block will terminate, in other words that loops will terminate [Elpa72].

The validity of program proving relies upon the notion that it is unlikely that a mistake will be made both in the program and a corresponding compensating mistake made in the assertions which are the basis of the proof. This is rather optimistic for all but the simplest programs. Program proving is now regarded as being too difficult to be of practical use.

2.3.1.4 Anomaly analysis

The first level of anomaly analysis is performed by the compiler to determine whether the program adheres to the language syntax. This first level of analysis is not usually considered testing. Testing is usually deemed to commence when a syntactically correct program is produced.

The second level of anomaly analysis searches for anomalies that are not outlawed by the programming language. Examples of such systems are DAVE [Oste76], FACES [Rama74] and TOOLPACK [Oste83]. Other systems such as SPADE[Carr86] and MALPAS[Webb87] also include anomaly analysis as a prerequisite to other system features. Anomalies which can be discovered by these systems include:

* the existence of unexecutable code (island code);
* problems concerning array bounds;
* failure to initialize variables;
* labels and variables which are unused;
* jumps into and out of loops;

and even:

* high complexity;
* departure from programming standards.

Discovery of these classes of problems is dependent on the analysis of the code. The first phase of anomaly analysis is to produce a flowgraph. This representation of the software can

now be easily scanned to identify anomalies. Determining infeasible paths is not within the bounds of anomaly analysis.

Some features of anomaly analysis have been grouped under the title data flow analysis. Here emphasis is placed on a careful analysis of the flow of data. Software may be viewed as flow of data from input to output. Input values contribute to intermediate values which in turn determine the output values. "It is the ordered use of data implicit in this process that is the central objective of study in data flow analysis" [Fosd76]. The anomalies detected are:

* Assigning values to a variable which is not used later in the program.
* Using a variable (in an expression or condition) which has not previously been assigned a value.
* (Re)assigning a variable without making use of a previously assigned value e.g.

```
10    x := 5
11    x := 10
```

Line 10 is redundant.

Data flow anomalies may arise from mistakes such as misspelling, confusion of variable names and incorrect parameter passing. The existence of a data flow anomaly is not evidence of a fault; it merely indicates the possibility of a fault. Software that contains data flow anomalies may be less likely to satisfy the functional requirements than software which does not contain them.

The role of data flow analysis is one of a program critic drawing attention to peculiar uses of variables. These peculiarities must be checked against the programmer's intentions and, if in disagreement, the program should be corrected.

2.3.2 Dynamic-functional

This class of technique executes test cases. No consideration is given to the detailed design of the software. The use of decision tables and cause-effect graphing creates test cases from the rules contained in the specification. Alternatively, test cases may be generated randomly. Equivalence partitioning [Myer79] creates test cases based on a decomposition of the required functions. Adaptive perturbation testing [Coop76, Andr81] attempts to create additional, more effective, test cases by modifying previous test cases. In all the approaches there is the need for an oracle to pronounce on the correctness of the output.

2.3.2.1 Equivalence partitioning

The aim of this technique is to devise test cases such that each case represents a set of equivalent test cases. The set of test cases form an equivalence class. The assumption is that if one test case in the equivalence class detects an error then all other test cases in the same class will also detect the same error. Equivalence partitioning is the technique of identifying the finite number of equivalence classes and devising a case to represent the class.

There are two types of equivalence class: valid and invalid. Valid equivalence classes are those that the software is required to process. Invalid equivalence classes are those that should

be rejected. It is important to devise test cases for both types of equivalence class.

Goodenough and Gerhart [Good75] suggest that "a basic hypothesis for the reliability and validity of testing is that the input domain of a program can be partitioned into a finite number of equivalence classes such that a test of a representative of each class will, by induction, test the entire class, and hence, the equivalent of exhaustive testing of the input domain can be performed".

2.3.2.2 Random testing

Random testing produces test data without reference to the code or the specification. The main software tool required is a random number generator. Duran and Natfos [Dura81, Dura84] describe how estimates of the operational reliability of the software can be derived from the results of random testing.

Potentially, there are some problems for random testing. The most significant is that it may seem that there is no guarantee of complete coverage of the program. For example, when a constraint on a path is an equality eg A=B+5 the likelihood of satisfying this constraint by random generation seems low. Alternatively, if complete coverage is achieved then it is likely to have generated a large number of test cases. The checking of the output from the execution would require an impractically high level of human effort.

Intuitively, random testing would appear to be of little practical value. Results from some

recent studies counter this view [Dura81, Dura84, Hekm86, Loo88]. Hekmatpoor, reporting on a series of experiments, states that a "striking result is that only a small fraction of the results of the full set of runs need to be examined in order to achieve the same degree of coverage as the full set of runs".

2.3.2.3 Adaptive perturbation testing

This technique is based on assessing the effectiveness of a set of test cases. The effectiveness measure is used to generate further test cases with the aim of increasing the effectiveness. Both Cooper and Andrews [Coop76, Andr81] describe systems which undertake this automatically for the testing of real-time systems.

The cornerstone of the technique is the use of executable assertions. The software developer inserts assertions into the software. The aim is to maximize the number of assertion violations. An initial set of test cases is provided by the tester. This is executed and the assertion violations recorded. Each test case is now considered in turn. The single input parameter of the test case that contributes least to the assertion violation count is identified. Optimization routines are used to find the best value to replace the discarded value such that the number of assertion violations is maximized. The test case is said to have undergone perturbation. This is repeated for each test case. The perturbed set of test cases are executed and the cycle is repeated until the number of violated assertions can be increased no further.

2.3.2.4 Decision tables and cause-effect graphing

The strength of this approach to test data selection lies in its exploration of the combinations of input values. Goodenough and Gerhart [Good75] and Myers [Myer79] both suggest the use of decision tables as a means of developing test cases based on the specification. Consider the following simple specification:

> The first character must be an "A" or a "B".
>
> The second character must be numeric.
>
> When these two conditions are satisfied update the file.
>
> When the first character is incorrect message M1 is given.
>
> When the second character is incorrect message M2 is given.

A decision table for representing the test cases required to test this specification is given in figure 2.4.

		1	2	3	4	5	6	7	8
c1	char1 = "A"	y	y	y	y	n	n	n	n
c2	char1 = "B"	y	y	n	n	y	y	n	n
c3	char2 numeric	y	n	y	n	y	n	y	n
70	make update			x		x			
71	message M1							x	x
72	message M2				x		x		x
	impossible case	x	x						

Figure 2.4 Decision table of test cases

The first two cases are impossible because the first character cannot be both "A" and "B" simultaneously and so can be discounted.

The construction of decision tables directly from specifications for large programs would result in exceedingly large decision tables containing many impossible cases. To avoid this Myers uses the cause-effect graph as a combinatorial logic network, rather like a circuit, making use of only the boolean logical operators AND, OR and NOT, as an intermediate notation between specification and decision table. Myers [Myer79] describes a series of steps for determining cases using cause-effect graphs and decision tables as follows:

* Divide the specification into workable pieces. A workable piece might be the specification for an individual transaction. This step is necessary because a cause-effect graph for a whole system would be too unwieldy for practical use.
* Identify causes and effects. A cause is an input stimulus, e.g. an input variable, an effect is an output response.
* Construct a graph to link the causes and effects in a way that represents the semantics of the specification. This is the cause-effect graph.
* Annotate the graph to show impossible effects and impossible combinations of causes.
* Convert the graph into a limited-entry decision table. Conditions represent the causes and actions represent the effects and rules (columns) represent the test cases.

Figure 2.5 shows a cause-effect graph for the example specification. Note the relationship marked 'e'. This shows a constraint between c1 and c2, in this case one of exclusivity. Only one of c1 and c2 can be true at one instant, though both may be false.

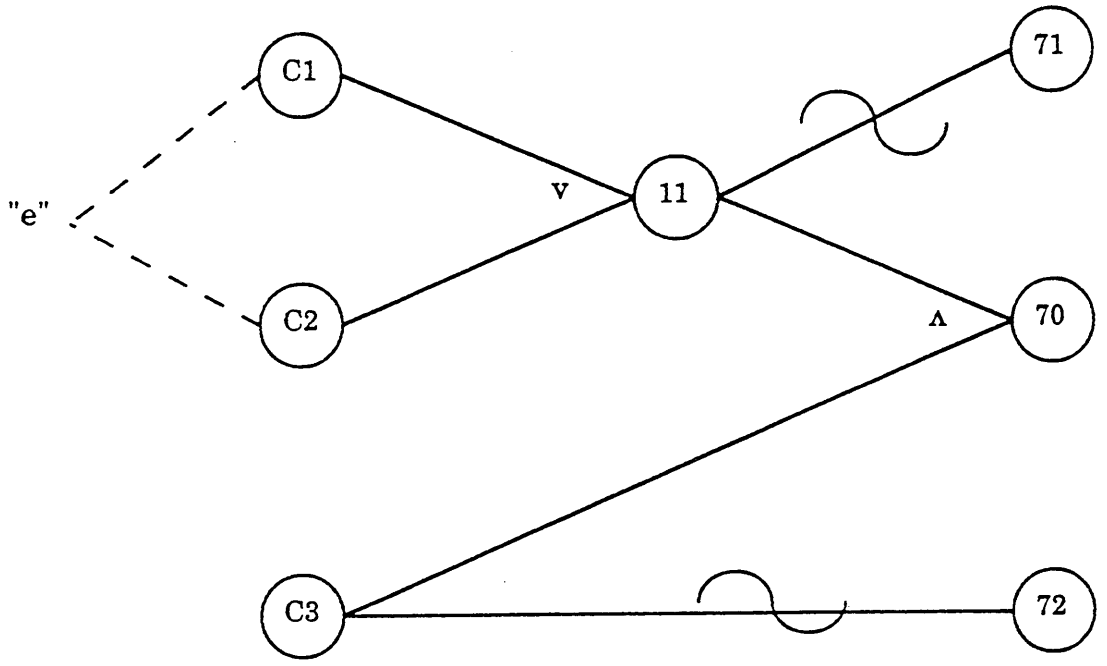

Figure 2.5 Cause-effect graph

Having constructed the cause-effect graph it is now used to construct a decision table. This is done by working from each effect and tracing back through the graph finding all combinations of causes that will yield the effect and all combinations that will not. Each combination requires a test to be included in the table. Consideration of the constraints bars the inclusion of impossible test cases.

In a simple case, such as the one above with 3 conditions, one may be tempted to feel that the cause-effect graph is an unnecessary intermediate representation. However, Myers illustrates the creation of test cases for a specification containing 18 causes. To progress immediately to the decision table would give 262,144 potential test cases. The purpose of the cause-effect graph is to identify a small number of useful test cases.

2.3.3 Dynamic-structural

Here the software is executed with test cases. Creation of the test cases is generally based upon an analysis of the software.

2.3.3.1 Domain and computation testing

Domain and computation testing are strategies for selecting test cases. They use the structure of the program and select paths which are used to identify domains. The assignment statements on the paths are used to consider the computations on the path. These approaches also make use of the ideas of symbolic execution.

A path computation is the set of algebraic expressions, one for each output variable, in terms of input variables and constants for a particular path. A path condition is the conjunction of constraints on the path. A path domain is the set of input values that satisfy the path condition. An empty path domain means that the path is infeasible and cannot be executed.

The class of error that results when a case follows the wrong path due to a fault in a conditional statement is termed a domain error. The class of error that results when a case correctly follows a path which contains faults in an assignment statement is termed a computation error.

"Domain testing is based on the observation that points satisfying boundary conditions are most sensitive to domain errors" [Clar83]. The domain testing strategy selects test data on and near the boundaries of each path domain [Whit80, Weuy80].

"Computation testing strategies focus on the detection of computation errors. Test data for which the path is sensitive to computation errors are selected by analyzing the symbolic representation of the path computation" [Clar83]. Clarke and Richardson list a set of guidelines for selecting test data for arithmetic and data manipulation computations.

2.3.3.2 Automatic test data generation

Use is made of automatic generation of test data when the program is to be executed and the aim is to achieve a particular level of coverage indicated by a coverage metric. One of the

earliest forms of automatic test data generator (ATDG) produced random values for input variables.

It has been suggested that test data can be generated from a syntactic description of the test data expressed in, say, BNF [Ince87]. This may seem novel as it is not usual to prepare such a syntactic description of the data, but it is a technique familiar to compiler writers [Bazz82, Hand70, Payn78]. In the case of compilers a carefully prepared data description, that of the programming language, is available. The principle may be transferable to test data generation in general.

Many ATDGs have used the approach of path identification and symbolic execution to aid the data generation process for example CASEGEN [Rama76] and the FORTRAN testbed [Hedl81]. The system of predicates produced for a path is part-way to generating test data. If the path predicates cannot be solved due to a contradiction, then the path is infeasible. Any solution of these predicates will provide a series of data values for the input variables so providing a test case.

Repeated use of the path generation and predicate solving parts of such a system may produce a set of test cases in which we have confidence of a high coverage of the program. The initial path generation will provide the highest coverage. Subsequent attempts to find feasible paths which incorporate remaining uncovered statements, branches and LCSAJs will prove increasingly difficult, some impossibly difficult.

A path-based approach which does not use symbolic execution is incorporated in the SMOTL system [Bice79]. The system has a novel approach to minimizing the number of paths required to achieve full branch coverage.

A program that has been tested with a high coverage may still not meet its specification. This may be due to the omission in the program of one of the functions defined in the specification. Data that is generated from the specification would prove useful in determining such omissions. To achieve this automatically requires a rigorous means of specification. The increasing use of formal specification methods may provide the necessary foundations on which to build automated functional test data generators.

2.3.3.3 Mutation analysis

Mutation analysis is not concerned with creating test data, nor of demonstrating that the program is correct. It is concerned with the quality of a set of test data [Budd78, Budd80]. Other forms of testing use the test data to test the program. Mutation analysis uses the program to test the test data.

The inclusion of mutation analysis as a dynamic-structural technique perhaps reduces the homogeneity of the class because it does not focus on how the cases were created but is concerned with assessing the quality of the test cases. Nevertheless, its inclusion in this class can be justified because it executes the software and makes some consideration of the software, in that a change is made to the software.

High quality test data will harshly exercise a program. To provide a measure of how well the program has been exercised, mutation analysis creates many, almost identical, programs. These programs are termed mutant programs. One change is made per mutant program. Each mutant program and the original program are executed with the same set of test data. The output from the original program is compared with the output from each mutant program in turn. If the outputs are different then that particular mutant is of little interest as the test data has discovered that there is a difference between the programs. This mutant is now 'killed' and disregarded. A mutant which produced output that matches with the original is interesting. The change has not been detected by the test data, and the mutant is said to be 'live'.

Once the output from all the mutants has been examined a ratio of killed to live mutants will be available. A high proportion of live mutants indicates a poor set of test data. A further set of test data must be devised and the process repeated until the number of live mutants is small, indicating that the program has been well tested.

A difficulty for mutation analysis occurs when a mutant program is an equivalent program to the original program. Although the mutant is textually different from the original it will always produce the same results as the original program. Mutation analysis will record this as a 'live' mutant even though no test data can be devised to kill it. The difficulty lies in the fact that determining the state of equivalence is, in general, unsolvable and hence cannot be taken into account when assessing the ratio of live to killed mutants.

Mutation analysis relies on the notion that if the test data discovers the single change that has been made to produce the mutant program then the test data will discover more major faults in the program. Thus, if the test data has not discovered any major faults and a high proportion of the mutants have been killed then the program is likely to be sound.

2.4 Effectiveness of program testing techniques

"If the use of a program testing technique is guaranteed to always reveal the presence of a particular error in a program, then the technique is said to be reliable for the error" [Howd78b]. Whilst much work has been undertaken to develop new approaches to testing, only a few researchers [Howd78b, Henn84, Basi87] have attempted to determine the effectiveness and reliability of existing program testing techniques.

2.4.1 Howden's study

One of the first pieces of empirical data on testing techniques is provided in a study by Howden [Howd78b]. The study tested six programs of various types using several different testing techniques. The techniques assessed were: path testing; branch testing; functional structured testing; integrated structured testing; special values testing; anomaly analysis; special requirements; interface analysis and symbolic testing.

The first five of these techniques are dynamic-structural approaches to testing. Path and branch testing are used here to mean executing all paths and all branches respectively at least once. Structured testing is an attempt to approximate to path testing which is usually

impossible because of the large, if not infinite, number of paths. Programs are decomposed into a hierarchy of functional modules and all paths through a module which require two or less iterations of each loop are tested at least once. Functional structured testing assesses an individual module while integrated structured testing treats each module in the context of the whole program. Special values testing is the act of testing the program with cases known to be problematic. For example, string processing programs are likely to exhibit errors in the handling of empty strings so a test case should contain empty strings.

The last four techniques are static approaches to testing. Path anomaly analysis is the examination of the program for suspicious looking features such as referencing a variable that has not been assigned a value. Special requirements testing involves checking that the specification stipulates the processing to be applied to all of the input domain dictated by the type of the input variables. Interface analysis is the checking of the consistency of the calling and called parameter lists. Symbolic testing symbolically executes paths. The output is used in two ways. First, it is used simply to generate expressions for the output variables and the path condition. Second, it is used to help generate test data which is then executed.

The generation of test cases and the static analyses were undertaken by hand but, because the use of the techniques require the application of well defined rules rather than the making of skill-based decisions the results are repeatable. The results of the analysis are given in figure 2.6.

Program	A	B	C	D	E	F	Total
Number of errors present	3	20	1	2	1	1	28
Number of errors found by method:							
1 Paths	2	12	1	2	1	0	18
2 Branches	3	3	0	0	0	0	06
3 Functional Structured	2	4	1	2	0	0	10
4 Integrated Structured	3	6	1	2	0	0	12
5 Combined Structured	3	6	1	2	0	0	12
6 Special Values	3	10	1	2	1	0	17
7 Anomaly Analysis	2	2	0	0	0	0	04
8 Special Requirements	0	4	1	2	0	0	07
9 Interface Analysis	0	2	0	0	0	0	02
10 Symbolic expressions	2	11	1	2	1	0	17
11 All (5-10)	3	19	1	2	1	0	26

Figure 2.6 Howden's analysis of testing techniques

Howden gives eight examples of errors to illustrate the differing combinations of techniques that will discover different classes of error. Figure 2.7 summarizes Howden's comments on the effectiveness of each technique for each of the eight example errors. A blank entry indicates that no comment was made about the technique's ability to discover the error, 'Y' indicates success and 'N' indicates that the technique was unable to discover the error.

Example error	1	2	3	4	5	6	7	8
1 Paths				N		N	N	
2 Branches	N			N		N	N	
3 Functional Structured				N		N	N	
4 Integrated Structured				N		N	N	
5 Combined Structured	Y	N	N	N	N	N	N	N
6 Special Values		Y		N	Y	N	N	
7 Anomaly Analysis			Y	N		N	N	
8 Special Requirements				Y		N	Y	
9 Interface Analysis				N		N	N	Y
10 Symbolic expressions				N	Y	Y	N	Y

Figure 2.7 Summary of Howden's analysis of effectiveness of testing techniques

Unfortunately, no mention is made of the two errors that could not be detected by any of the techniques. Insufficient data is provided to establish which combinations of the techniques are sufficient on their own to discover most of the errors.

The results are encouraging for the use of symbolically executed expressions for output variables. Out of a total of 28 errors five were discovered where it would be "possible for the incorrect variable to take on the values of the correct variable during testing on actual data, thus hiding the presence of the error" [Howd78b]. The paper concluded that the testing strategy most likely to produce reliable software was one that made use of a variety of techniques. Unfortunately, no particular set of techniques was proposed, and it was suggested that further research is required to determine the best combination of techniques for particular circumstances.

2.4.2 Hennell, Hedley and Riddell's study

Hennell et al [Henn84] undertook a study to assess the effectiveness of the LDRA testbed in finding faults in an already working system which has previously been tested by ad hoc methods. In the two experiments the LDRA testbed discovered errors not discovered by previous testing.

The testbed measures three coverage metrics TER1, TER2 and TER3 which show the percentage of statements, branches and LCSAJs respectively that have been covered by the testing.

Figures 2.8a and 2.8b show the coverage of statements, branches and LCSAJs attained firstly (TD1) by the best functional test data and then showing how other tests were added (TD2, TD3...) to give acceptable cover. These tests found 33 errors in the two programs not detected

by previous ad hoc testing. Figure 2.9 shows the errors classified into 13 fault classes. Unfortunately, but perhaps not surprisingly, the authors experienced difficulty in attributing the detection of the errors to TER1, TER2 and TER3.

	TER1 %	TER2 %	TER3 %	
TD1	78.6	63.6	40.6	Functional test data set
TD2	96.4	88.9	64.0	
TD3	97.8	91.3	65.8	
TD4	98.0	92.6	67.1	
TD5	98.6	93.0	67.4	
TD6	99.0	93.8	68.0	
TD7	99.4	94.7	68.6	
TD8	100.0	97.1	71.7	
TD9	100.0	97.5	72.4	

Figures 2.8a Coverage for a COBOL program

	TER1 %	TER2 %	TER3 %	
TD1	61.5	47.9	37.5	Functional test data set
TD2	73.5	63.5	50.7	
TD3	78.2	68.9	55.4	
TD4	83.6	76.5	62.5	
TD5	88.0	81.5	70.0	
TD6	90.9	87.0	75.1	
TD7	91.3	87.4	76.1	

Figures 2.8b Coverage for a PL/1 program

1.	Initialization error	2
2.	Single statement error	3
3.	Two uncoupled single statement errors	1
4.	Missing loop	2
5.	Compensating loop errors	2
6.	Incorrect predicate	1
7.	Compensating loop and alternates	1
8.	Alternate missing in nested loops	2
9.	Alternate missing in nested alternates in a loop (or loops) in nested alternates	6
10.	Loop missing in nested alternates in a loop	1
11.	Combinatorial error from nested alternates in nested loop alternates	3
12.	Combination between nested loops	1
13.	None of the above	5

Figure 2.9 Analysis of errors

The authors conclude that after many years in attempting to quantify the testing process, satisfying TER3 = 1 in conjunction with a functional test set is the most cost-effective method of finding errors. However, explaining why this technique is so effective is not easy.

2.4.3 Basili and Selby's study

Basili and Selby's study [Basi88] involved experiments to determine how testing effectiveness relates to several factors: testing technique; software type; fault type; tester experience; and any interaction among these factors.

The testing techniques examined are: functional testing, in particular equivalence class partitioning and boundary value analysis; structural testing using 100% statement coverage; and code reading by stepwise extraction.

The study examines three different aspects of software testing: fault detection effectiveness; fault detection cost; and classes of fault detected. The framework for the study is summarized in figure 2.10.

Four programs are used in the experiment. Each one a different type: P1 a text formatter; P2 undertakes mathematical plotting; P3 implements a numeric abstract data type; and P4 a database maintainer. They contain a total of 34 faults distributed between the four programs, 9, 6, 7 and 12 respectively. The faults can be classified in many ways. The authors used two classifications and the distribution of the errors between them is given in figure 2.11.

I. Fault detection effectiveness

(a) For programmers doing unit testing, which of the testing techniques detects the most faults in programs ?

1. Which of the techniques detects the greatest percentage of faults in the programs ?

2. Which of the techniques exposes the greatest percentage of program faults (faults that are observable but not necessarily reported) ?

(b) Is the number of faults observed dependent on software type ?

(c) Is the number of faults observed dependent on the expertise level of the person testing?

II. Fault detection cost

(a) For programmers doing unit testing, which of the testing techniques detects the faults at the highest rate ?

(b) Is the fault detection rate dependent on software type ?

(c) Is the fault detection rate dependent on the expertise level of the person testing?

III. Classes of fault observed

(a) For programmers doing unit testing, do the methods tend to capture different classes of faults ?

(b) What classes of faults are observable but go unreported?

Figure 2.10 Outline of goals and questions for testing experiment [Basi87]

	Omission	Commission	Total
Initialization	0	2	2
Computation	4	4	8
Control	2	5	7
Interface	2	11	13
Data	2	1	3
Cosmetic	0	1	1
Total	10	24	34

Figure 2.11 Distribution of faults in the program [Basi87]

The experimental design used in the final phase of the experiment is a fractional factorial analysis of variance design. This allows assessment of the three main effects: testing technique; software type; and level of expertise, and assessment of the interactions between

these effects. The design also measured several dependent variables: percentage of faults detected; time taken; and from the on-line testing various measures such as number of executions of programs.

The following is the authors' summary of the results:

1. With the professional programmers, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate. (By fault detection rate the author means the number of faults discovered divided by the time spent looking for them.)

2. In one group of junior and intermediate level programmers, code reading and functional testing were not different in faults found, but were both superior to structural testing, while in another similar group there was no difference among the techniques.

3. With the junior and intermediate levels, the three techniques were not different in fault detection rate.

4. Number of faults observed, fault detection rate and total effort in detection depended on the type of software tested.

5. Code reading detected more interface faults than did the other methods.

6. Functional testing detected more control faults than did the other methods.

7. When asked to estimate the percentage of faults detected, code readers gave the most accurate estimates while functional testers gave the least accurate estimates.

The authors' also report that "there were few significant interactions between the main effects of testing technique, program and expertise level".

2.4.4 Summary of empirical data on effectiveness of testing techniques

The main conclusion arising from analyzing the empirical data that exists is that use of a variety of testing techniques is likely to be more effective than use of a single technique. However, it is not clear whether all techniques need to be used to discover all faults or whether there are sets of compatible techniques which, when used together, are likely to discover all of the faults.

The results produced so far are also limited to a few comparatively small studies. These need validating by many more repeated studies along similar lines.

2.5 Further Evaluation of Testing Techniques

2.5.1 Classifying evaluation techniques

Approaches to the evaluation of testing techniques may be ordered according to the degree of observation, experimentation, and theoretical evaluation involved. At one end of the scale lies a theoretical approach that makes no use of sample programs nor of human testers. In the middle sits the laboratory experiment involving many well understood sample programs and many people whose abilities are known. At the other end is the observational study of real situations over a long period. The three studies described above are at different points on the theoretical-experimental-observational spectrum.

The study by Howden is the nearest to the theoretical end of the spectrum. No human testers were involved. A theoretically perfect-application of the techniques under evaluation was undertaken by the evaluator. Six programs with known errors were used. Had a classification of errors been used to determine which of the techniques is capable of detecting each error then the approach would have been entirely theoretical.

The study by Basili and Selby is the closest to the experimental midpoint. Many human testers were set the task of testing four quite different programs each containing a set of known errors.

The study by Hennell, et al is the nearest of the three to the observational end. Two

production programs were tested by regular users of a testing tool. This study is however a long way from the end of the spectrum as the testing was not undertaken as a normal part of the software's development but as a special experiment.

The main shortcoming of the studies undertaken to date is the small number of them and, as a result, the shortage of data. Basil and Selby's study appears to be a sound experimental design but it avoids the thorny problem of the differences between the laboratory experiment and the live software development environment. Ideally, this study should have been expanded to include a larger number of software testers; assessing more 'real' programs and covering more testing techniques. This would provide valuable information about which combination of techniques would be the most appropriate for use in testing particular types of software with staff of particular levels of experience. Such a study would of course take a long time and be very expensive.

2.5.2 Proposals for further evaluation of testing techniques

Three approaches are proposed for consideration for the further evaluation of testing techniques: observation; experiment; and theory.

An observational study requires the researcher to identify, for each technique to be assessed, several organizations that currently use the technique as part of their normal software development. Over a long period the development of software is to be monitored and details

of every fault discovered at every stage of the life of the software are to be recorded. The aim of the analysis is to assess the number of errors discovered during development using the techniques under investigation and to compare this to the number of errors detected later in the software product's life. This will tell us which technique detects the most errors early in the life of the software. By recording details of software type, experience of user and effort expended assessment can be made of the most cost-effective method.

The experimental study should follow the same experimental design as the study by Basili and Selby. It should assess more testing techniques using more testers from a wider variety of organizations. The main aim of this study would be to gather more data to corroborate the findings of Basili and Selby. Indeed, there would be much to be gained from simply repeating their experiment with different subjects in other environments.

A theoretical study would commence by establishing or selecting a classification of software faults such as those proposed by Van Tassel [VanT78] and Kamer [Kame88]. A set of programs is then constructed or identified which contain all the faults listed in the classification. The evaluation now assesses which of the techniques discover the existence of which faults. A weakness of this approach is that it would be difficult to assess cost-effectiveness of the techniques.

2.6 Summary

The principal objective of software testing is to gain confidence in the software. This necessitates the discovery of both errors of omission and commission. Confidence arises from thorough testing. There are many testing techniques which help to achieve thorough testing.

Testing techniques can be assessed according to where along the two main testing strategy dimensions they fall. The first dimension, the functional-structural dimension, assesses the extent to which the function description in the specification, as opposed to the detailed design of the software, is used as a basis for testing. The second dimension, the static-dynamic dimension, considers the degree to which the technique executes the software and assesses its run-time behaviour as opposed to inferring its run-time behaviour from an examination of the software. These two dimensions can be used to produce four categories of testing technique: static-functional, static-structural, dynamic-functional and dynamic-structural. As with all classifications this one is problematic at the boundaries. Some techniques appear to belong equally well in two categories.

The aims of the testing techniques range from demonstrating correctness for all input classes (e.g. program proving), to showing that for a particular set of test cases no faults were discovered (e.g. random testing). Debate continues as to whether correctness can be proved for life-size software and about what can be inferred when a set of test cases finds no errors. A major question facing dynamic testing techniques is whether the execution of a single case demonstrates anything more than that the software works for that particular case. This has led

to work on the identification of domains leading to the assertion that a test case represents a particular domain of possible test cases.

Many of the structural techniques rely on the generation of paths through the software. These techniques are hampered by the lack of a sensible path generation strategy. There is no clear notion of what constitutes a 'revealing' path worthy of investigation as opposed to a 'concealing' path which tells the tester very little.

Testers often utilize their experience of classes of faults associated with particular functions and data types to create additional test cases. To date there is no formal way of taking account of these heuristics.

A significant feature of the little empirical data that has been collected is that use of a variety of testing techniques is likely to detect more errors than reliance upon a single technique. Over the last few years effort has been directed at construction of integrated, multi-technique software development environments.

Symbolic execution looks to be a promising technique. In an experiment Howden [Howd78b] discovered that symbolic execution was able to discover faults that other techniques would have missed. Yet, few full symbolic execution systems currently exist. Of the experimental systems that have been developed none addresses commercial data processing software written in languages such as COBOL.

CHAPTER THREE PRINCIPLES OF SYMBOLIC EXECUTION

Symbolic execution creates a set of values for the input variables to a program. The novelty is in the nature of the values. Rather than create a set of actual values a set of symbolic values are produced.

The following program fragment determines netpay.

```
accept grosspay
accept taxpcent
accept taxfree
compute taxable = grosspay - taxfree
compute netpay = grosspay - (taxable * taxpcent/100)
display netpay
```

For each of the input variables the following symbolic values may be used as input.

Input Variable	Symbolic value
grosspay	g
taxpcent	pc
taxfree	tf

After symbolically executing the program the value of netpay would be :

```
g-(g-tf)*pc/100
```

This expression holds for say the numeric values of 600, 25 and 200 for g, pc and tf respectively, but it also represents far more sets of values than that single case. The expression represents a domain of test cases.

3.1 Symbolic evaluation

Clarke and Richardson [Clar81] use the phrase 'symbolic evaluation' as a collective title for techniques that make use of algebraic expressions to represent the values of variables. This use is not universal and the phrase may be used as a synonym for symbolic execution. There

are three types of symbolic evaluation as defined by Clarke and Richardson: symbolic execution; dynamic symbolic evaluation; and global symbolic evaluation.

3.2 Symbolic execution

This is the evaluation of a single path using symbolic values. The output from the symbolic execution has two components. First, a set of expressions in terms of the symbolic values of the input variables and constants. Second, a set of constraints which must be satisfied in order for the path to be executable. Collectively these constraints are known as the path condition. There are two approaches to deriving the symbolic expressions and path condition: forward expansion and backward substitution. These are described later in this chapter.

3.3 Dynamic symbolic evaluation

Here symbolic execution takes place in parallel with the execution of actual value cases. For each variable encountered both the actual value and a symbolic expression are maintained. No feasibility checking of the path condition is required as the execution of the actual value case ensures that the path followed is executable unless a run time error is produced.

Dynamic symbolic evaluation is applied principally in debugging. In addition, user inserted assertions may be checked to see whether the processing of the input case complies with the assertion at that point in the path.

3.4 Global symbolic evaluation

The aim of global symbolic evaluation is to derive a representation for a whole module. Whereas symbolic execution produces a representation of a single path in terms of its input variables and constants, global symbolic evaluation aims to achieve representations for all paths through a routine.

Where a routine contains only one path (no conditional statements) the result is identical to symbolic execution. When conditional statements are present then a set of path conditions and variable expressions are produced. When a routine contains loops then there is potentially an infinite number of paths. Global symbolic evaluation aims to identify a loop expression. From this further expressions representing successive iterations may be derived providing recurrence relations in terms of the values of variables at successive iterations.

The next step involves a loop analysis to solve the recurrence relations. As a result many paths representing differing numbers of iterations of a loop are represented by a single expression. Clarke and Richardson state that "this is not always straightforward and sometimes may not be possible. .... In particular, the dependence may be cyclic -- V may be mapped on W, which depends on V -- in which case the recurrence relations cannot be solved". Howden [Howd78b] demonstrates that in general such recurrence relations cannot be solved, in which case global symbolic evaluation has a bleak future.

Of these three classes of symbolic evaluation this thesis addresses symbolic execution.

3.5 Flowgraph

Before considering symbolic execution of a path it is useful to introduce the flowgraph representation of a program. The flowgraph is a useful way of displaying the program's structure. It consists of 'nodes' joined by 'arcs'. The nodes represent branch points within the program and the arcs represent the statements that modify variables. It is easier to identify a particular path from a flowgraph than from the source code of the program.

The program in figure 3.1 accepts three integers that are interpreted as lengths of the sides of a triangle. It determines the type of triangle that the values represent and calculates the area of the triangle. The program can be represented by the flowgraph in Figure 3.2. A node on the graph corresponds to a selection statement in the program and an arc on the graph corresponds to a series of input, assignment and output statements.

The flowgraph contains 33 paths. Symbolic execution considers a single path at a time. There are two approaches to symbolic execution of a path: forward expansion and backward substitution.

3.6 Forward expansion

Forward expansion is similar to the normal execution of a path. A path consists of a series of input statements, condition predicates and assignment statements. The symbolic execution commences at the root of the path and proceeds branch by branch to the end of the path. During a path traversal each input variable is given a symbol rather than an actual value.

```
identification division.
program-id. triangle.
data division.
working-storage section.
01  I       pic 9.
01  J       pic 9.
01  K       pic 9.
01  A       pic 99.
01  B       pic 99.
01  C       pic 99.
01  S       pic 99.
01  match   pic 9.
01  area    pic 999.
begin-triangle.
     accept I,J,K
     if I + J > K and J + K > I and K + I > J
     then
       move 0 to match
       compute S = (I + J + K) / 2
       compute A = S - I
       compute B = S - J
       compute C = S - K
       compute area = (S * A * B * C) ** 0.5
       if I = J
       then
         add 1 to match
       end-if
       if J = K
       then
         add 1 to match
       end-if
       if K = I
       then
         add 1 to match
       end-if
       evaluate true
         match = 0      display 'Scalene'
         match = 1      display 'Isosceles'
         match = 3      display 'Equilateral'
         other          display 'Error'
       end-evaluate
       display area
     else
       display 'Not a Triangle'
     end-if
     stop run
end program triangle.
```

Figure 3.1 Triangle program

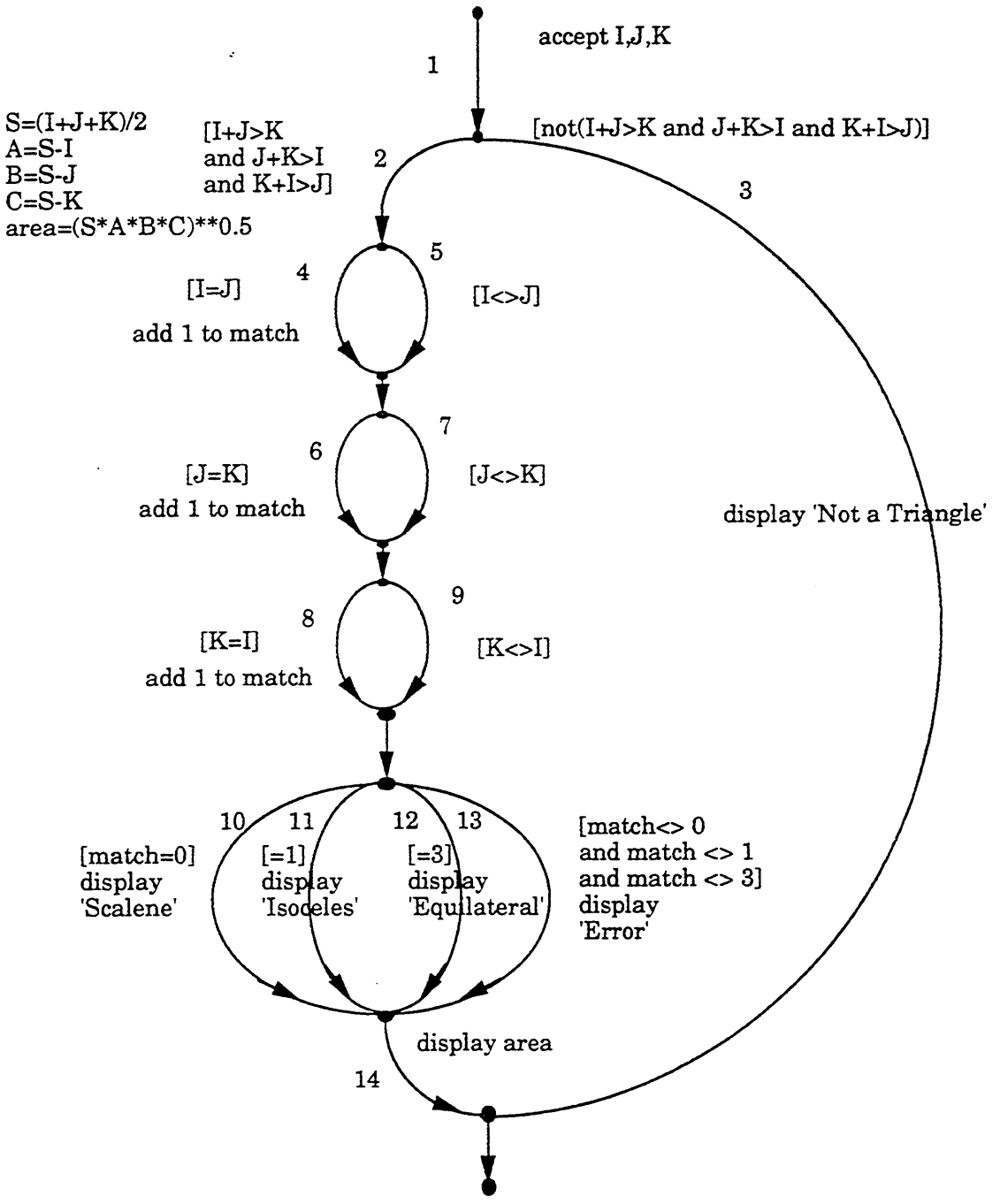

Figure 3.2 Flowgraph for the triangle program

Thereafter, each assignment statement is evaluated so that it is expressed in terms of symbolic values of input variables and constants.

At the end of the symbolic execution of a path, the output variables will be represented by expressions in terms of symbolic values of input variables and constants. The output expressions are subject to constraints. A list of these constraints, known as the Path Condition (PC), is provided by the set of symbolic representations of each condition predicate along the path. Analysis of these constraints may indicate that the path is not executable due to a contradiction.

For each path, a path condition is maintained together with a table containing each variable referenced and a corresponding expression. Initially, each variable expression is empty. As symbolic execution proceeds along a path four categories of statement are identified: input; assignment; predicate; and output.

When an input statement is encountered a new symbolic value is created and replaces the current expression representing the variable.

When an assignment is encountered two substitutions take place. First, the variables in the right hand side of the assignment statement are substituted by their current expressions. Second, the resulting assignment expression is used to update the expression representing the target variable. For example, when the assignment is COMPUTE A = B + C. First, replace

B and C by their current expressions say X + Y and J + K giving COMPUTE A = X + Y + J + K. Second, replace the current expression for A by the newly created expression.

When a predicate is encountered each variable in the predicate is substituted by its current expression. Next the predicate is tested for truth. When true it is ignored because it has no impact on the PC e.g. A = A. When false the symbolic execution halts because a contradiction has been discovered e.g. A <> A. When its truth is indeterminate e.g. A = B then the predicate is conjoined onto the PC.

When an output is encountered the current expression is displayed.

To illustrate symbolic execution by forward expansion consider the path covering branches (arcs) 1, 2, 4, 6, 8, 12 and 14. The branches can be written in sequence to isolate the path into a straight-line form, where predicates are shown in square brackets, as follows:

```
1    accept I J K
2    [I+J >K and J+K >I and K+I >J]
     move 0 to match
     compute S = (I+J+K)/2
     compute A = S-I
     compute B = S-J
     compute C = S-K
     compute area = (S*A*B*C)**0.5
4    [I=J]
     add 1 to match
6    [J=K]
     add 1 to match
8    [K=I]
     add 1 to match
12   [match = 3]
     display 'Equilateral'
14   display area
```

Symbolic execution of this path is now described. The current expression for a variable is shown as:

variable: expression

e.g. total: 0 indicates that the current expression for total is 0. The PC is shown in a similar way.

1 accept I J K

This is an input statement so the expressions for I, J and K are given new symbolic values.

I: i, J: j, K: k

2 [I+J >K and J+K >I and K+I >J]

To progress beyond this point the test case must satisfy the predicate enclosed within the brackets. First, substitute I, J and K by their current expressions.

i+j >k and j+k >i and k+i >j

Second, decide whether its truth can be determined - no.

Third, conjoin to the PC.

PC: i+j>k and j+k>i and k+i>j

move 0 to match

This is an assignment so the expression for match is replaced by 0.

match: 0

compute S = (I+J+K)/2

When a variable is assigned the value of an expression the variables that are incorporated in the expression are substituted by their current symbolic expression. In the case of I, J and K their current symbolic values are i, j and k.

S: (i+j+k)/2

compute A = S-I

The symbolic value of A is achieved by substituting the variables on the right-hand-side by their current expression.

A: (i+j+k)/2-i

compute B = S-J

B: (i+j+k)/2-j

compute C = S-K

C: (i+j+k)/2-k

compute area = (S*A*B*C)**0.5

area: (((i+j+k)/2)*((i+j+k)/2-i)*((i+j+k)/2-j)*(i+j+k)/2-k))**0.5

4 [I=J] i=j is conjoined to PC.

PC: i+j>k and j+k>i and k+i>j and i=j

The PC may be simplified (but this is not essential).

PC: i=j and 2*i>k and k>0

add 1 to match

match is incremented by 1.

match: 1

6 [J=K] j=k is conjoined to PC.

PC: i=j and 2*i>k and k>0 and j=k

simplifying gives:

PC: i=j and j=k and k>0

add 1 to match

match: 2

8 [K=I] k=i is conjoined to the PC.

PC is i=j and j=k and k>0 and k=i

Simplifying merely removes the added constraint k=i.

PC is i=j and j=k and k>0

add 1 to match

match: 3

12 [match = 3] Substitute match with its current expression.

3 = 3 This is true ignore predicate.

display 'Equilateral'

Note the output string.

14 display area

Note the output expression.

At the end of symbolic execution of the path the following inputs and outputs have been identified.

Inputs

I = i
J = j
K = k

Output

'Equilateral'
area = (((i+j+k)/2)*((i+j+k)/2-i)*((i+j+k)/2-j)*(i+j+k)/2-k))**0.5
PC is i=j and j=k and k>0

The path condition dictates that output of the string 'Equilateral' will result when i=j=k>0. This matches with expectation. It is not so straightforward to verify that the output for 'area' is correct. This is most easily achieved by substituting i, j and k with values, evaluating the expression and comparing the result against an earlier calculation of the result for the chosen values.

The path considered above is feasible but some paths chosen through a program are likely to be infeasible. As an illustration of symbolically executing an infeasible path by forward expansion consider the path 1, 2, 4, 7, 8, 13 and 14.

The branches of this path are as follows:

```
1    accept I J K
2    [I+J >K and J+K >I and K+I >J]
     move 0 to match
     compute s = (I+J+K)/2
     compute a = S-I
     compute b = S-J
     compute c   S-K
     compute area = (S*A*B*C)**0.5
4    [I=J]
     add 1 to match
7    [J<>K]
8    [K=I]
     add 1 to match
13   [match <> 0 and match <> 1 and match <> 3]
     display 'Error'
14   display area
```

Symbolic execution of this path follows.

1 accept I J K

I: i, J: i, K: k

2 [I+J >K and J+K >I and K+I >J]

PC: i+j>k and j+k>i and k+i>j

move 0 to match

match: 0

compute s = (I+J+K)/2

S: (i+j+k)/2

compute a = S-I

A: (i+j+k)/2-i

compute b = S-J

B: (i+j+k)/2-j

compute c = S-K

C: (i+j+k)/2-k

compute area = (S*A*B*C)**0.5

area: (((i+j+k)/2)*((i+j+k)/2-i)*((i+j+k)/2-j)*(i+j+k)/2-k))**0.5

4 [I=J]

PC: i+j>k and j+k>i and k+i>j and i=j

Simplifying to

PC: i=j and 2*i>k and k>0

add 1 to match

match: 1

To this point the symbolic execution is the same as for the path described earlier.

7 [J<>K]

PC: i=j and 2*i>k and k>0 and j <> k

8 [K=I]

PC: i=j and 2*i>k and k>0 and j<>k and k=i

Simplifying to

PC: i=j and j=k and j<>k and k>0

This PC is infeasible. It is not worthwhile continuing the symbolic execution further as it will remain infeasible.

This approach to symbolic execution is termed 'forward expansion' because the execution commences from the root of the directed graph and progresses through the program from the

entry point to an exit point. As the traversal of the path progresses the Path Condition is gradually expanded as each condition and relevant variable assignment is encountered. It has the advantage of detecting infeasibility at the point at which the infeasibility first occurs.

3.7 Backward substitution

Backward substitution adopts the opposite approach to forward expansion. It starts the symbolic execution of a path from the terminating point of the program as opposed to the starting point.

Backward substitution of a path is a more complicated procedure than forward expansion. It is iterative in nature repeatedly returning to expressions previously processed. Consider again the feasible path 1, 2, 4, 6, 8, 12 and 14 which is now symbolically executed backwards.

14 display area

At this point no values have been attributed to the variable area but it is important to note the occurrence of each output variable.

display 'Equilateral'

Note the output of the string.

12 [match = 3]

PC: match = 3

add 1 to match

As the initial value (symbolic or actual) is not known at this stage the expression must remain in terms of match.

match: match + 1

8 [K=I]

PC: match = 3 and K = I

add 1 to match

Match has already been encountered at branch 12 resulting in an expression which must now be modified. This is achieved by substituting the expression created at branch 12 into the occurrence of match on the RHS of the assignment in branch 8. Thus 'match' is substituted by 'match + 1' giving 'match + 1 + 1' hence: match: match + 2

6 [J=K]

PC: match=3 and K=I and J=K

add 1 to match

match: match + 3

4 [I=J]

PC: match=3 and K=I and J=K and K=I

simplifies to

PC: match=3 and K=I and J=K

compute area = (S*A*B*C)**0.5

area: (S*A*B*C)**0.5

compute C = S-K

C: S-K

The handling of this assignment illustrates the naming of backward substitution. At each assignment a search must be made through each expression of each previously noted variable for the variable being assigned the new value. When the variable is found in an expression it is substituted by the new expression resulting from the assignment. In this case the previous expression for the variable area must now be modified to take account of the new value of variable C.

area: (S*A*B*(S-K))**0.5

compute B = S-J

B: S-J

area: (S*A*(S-J)*(S-K))**0.5

compute A = S-I

A: S-I

area: (S*(S-I)*(S-J)*(S-K))**0.5

compute S = (I+J+K)/2

S: (I+J+K)/2

Again area must be modified so too must A, B and C.

area: (((I+J+K)/2)*((I+J+K)/2-I)*((I+J+K)/2-J)*(I+J+K)/2-K))**0.5

A: (I+J+K)/2-I

B: (I+J+K)/2-J

C: (I+J+K)/2-K

move 0 to match

Current expression for match is

match: match + 3

Substitute 0 in place of match in the expression.

match: 3

PC: 3=3 and K=I and J=K

simplifies to

PC: K=I and J=K

2 [I+J >K and J+K >I and K+I >J]

PC: K=I and J=K and I+J>K and J+K>I and K+I>J

which simplifies to give:

PC: K=I and J=K and I>0

1 accept I J K

I: i, J: j, K:k

area: (((i+j+k)/2)*((i+j+k)/2-i)*((i+j+k)/2-j)*(i+j+k)/2-k))**0.5

A: (i+j+k)/2-i

B: (i+j+k)/2-j

C: (i+j+k)/2-k

The backward substitution approach to symbolic execution yields exactly the same results as forward expansion described above.

The next illustration is of backward substitution of an infeasible path. Consider again the path 1, 2, 4, 7, 8, 13 and 14.

14 display area

Note output variable.

display 'Error'

Note output string.

13 [match <> 0 and match <> 1 and match <> 3]

PC: match <> 0 and match <> 1 and match <> 3

add 1 to match

match: match + 1

8 [K=I]

PC: match <> 0 and match <> 1 and match <> 3 and K=I 7 [J<>K]

PC: match <> 0 and match <> 1 and match <> 3 and K=I and J<>K

add 1 to match

match: match + 2

4 [I=J]

PC: match <> 0 and match <> 1 and match <> 3 and K=I and J<>K and I=J

At this point the path has become infeasible because J cannot be both equal to K and not equal to K simultaneously.

Ramamoorthy [Rama76] claims that in backward substitution assignment statements that do not affect any conditional statements need not be symbolically executed. This is not wholly satisfactory. Application of this rule could cause the omission of assignments that affect the expressions of output variables. Four categories of assignment statement can be identified:

1. assignments that affect conditional statements and output variables;
2. assignments that affect only conditional statements;
3. assignments that affect only output variables;
4. assignments that affect neither conditional statements nor output variables.

By application of Ramamoorthy's approach categories (1) and (2) are included in the symbolic execution whilst (3) and (4) are excluded. It is the omission of category (3) that is of concern. The set of output variable expressions would be incomplete even though the PC would be accurate. An occurrence of category (4) in a program is of interest. If an assignment affects neither output variables nor the PC then its presence in the program should be questioned.

Ramamoorthy [Rama76] claims that backward substitution has the benefit of being the simpler operationally and requires the least storage space. However, forward expansion may determine infeasibility more quickly, handles arrays more simply and perhaps most importantly is intuitively more obvious than backward substitution. It is these advantages that have led to the more widespread adoption of forward expansion at the expense of backward substitution.

3.8 Simplification of symbolic expressions

The usefulness of the output of symbolic expressions, though potentially helpful, is hampered by the unfamiliar format of the expressions. One possible solution to this difficulty is to attempt to simplify the expression in the hope that it will be more meaningful. By simplifying an expression evidence is lost of how the expression was calculated. This evidence may prove helpful in pin-pointing a fault in the program. Both the simplified and the unsimplified expressions should be provided as output.

In the above examples of the output from feasible paths the expression for area is expressed as:

```
area = (((i+j+k)/2)*((i+j+k)/2-i)*((i+j+k)/2-j)*(i+j+k)/2-k))**0.5
```

This expression could be simplified to:

```
area = ((i+j+k)*(-i+j+k)*(i-j+k)*(i+j-k))**0.5/4
```

In algebraic terms these two expressions are equivalent. A potential difficulty exists when expressions are evaluated using particular machines. Execution of the two expressions should give identical outputs but rounding and truncation of intermediate results may yield slightly different results. This may cause execution of an unintended path to take place.

3.9 Problems in applying symbolic execution

There are four problem areas which are well documented in the literature concerning the application of symbolic execution. These are: path selection and the evaluation of loops; a

dilemma over how to process module calls; the evaluation of array references dependent on input values; and the checking of path feasibility. There are three further problems concerning the application of numerical optimizers for checking feasibility which are not documented in the literature; these are described in Part Two.

3.9.1 Path selection and loops

Current methods of path selection employ only simple strategies such as take the true branch first or generate the shortest path. These strategies have a common target, that of achieving a particular coverage metric such as all statements or all branches are executed at least once. Each of these strategies is prey to the problem of selecting infeasible paths. Having identified a set of paths which cover say, all branches, some of the selected paths will be found to be infeasible. This leaves the problem of identifying feasible paths which include the non-covered branches resulting from the infeasible paths. Further, such a strategy does not consider the usefulness of the selected paths.

Many systems that use symbolic execution have two distinct stages. First, select a path. Second, symbolically execute the selected path. Ideally, a symbolic execution testing system should incorporate a path selection strategy in which path selection and symbolic execution take place together. Such co-operative processing allows the expressions produced during symbolic execution to be utilised in path selection in an attempt to select only feasible paths.

Symbolic execution cannot proceed beyond a loop unless the number of iterations is known. A common strategy for placing loops on paths is to create three paths: one that contains zero iterations of the loop; a second that contains one iteration; and a third containing two iterations. The idea behind this strategy is that zero, one and two iterations coincide with three classes of loop error and so each should be tested. Consider the following loop:

```
perform until x = y
  compute x = x + 1
  display "-" no advancing
end-perform
```

Assume that x and y are input variables and that earlier processing has guaranteed that x is less than or equal to y. Three paths would be selected where the following predicates are conjoined to the path condition.

Path Number	Number of iterations	Predicates conjoined to Path Condition
1	0	x = y
2	1	x < y, x + 1 = y
3	2	x < y, x + 1 < y, x + 2 = y

These three paths would provide reasonable coverage of the loop.

Consider the following code which produces the Fibonacci series. The program accepts three values; the first two act as commencing values and the third as the number of terms to be displayed.

```
1   accept last this A
2   move 2 to N
3   display last
4   display this
5   perform until N >= A
6     compute next = last + this
7     move this to last
8     move next to this
9     display next
10    add 1 to N
11  end-perform
```

The important feature of this program for symbolic execution is that the number of iterations

of the loop is controlled by the input variable A. To demonstrate the impact of this feature consider the following symbolic execution of the program where the symbolic expressions for the output variables and the state of the PC are given in the right-hand column.

```
1   accept last this A            A: a, last: 1, this: t
2   move 2 to N                   N: 2
3   display last                  Output  1
4   display this                  Output  t
5   [ N >= A ]                    PC: 2 >= A
6   compute next = last + this    next: 1 + t
7   move this to last             last: t
8   move next to this             this: 1 + t
9   display next                  Output  1 + t
10  add 1 to N                    N: 3
11  end-perform
5   [ N >= A ]                    PC: 3 >= A
6   compute next = last + this    next: 1 + 2t
7   move this to last             last: 1 + t
8   move next to this             this: 1 + 2t
9   display next                  Output  1 + 2t
10  add 1 to N                    N: 4
11  end-perform
5   [ N >= A ]                    PC: 4 >= A
6   compute next = last + this    next: 21 + 3t
7   move this to last             last: 1 + 2t
8   move next to this             this: 21 + 3t
9   display next                  Output  21 + 3t
10  add 1 to N                    N: 5
11  end-perform
5   [ N >= A ]                    PC: 5 >= A
6   compute next = last + this    next: 31 + 5t
7   move this to last             last: 21 + 3t
8   move next to this             this: 31 + 5t
9   display next                  Output  31 + 5t
10  add 1 to N                    N: 6
11  end-perform
5   [ N >= A ]                    PC: 6 >= A
6   compute next = last + this    next: 51 + 8t
7   move this to last             last: 31 + 5t
8   move next to this             this: 51 + 8t
9   display next                  Output  51 + 8t
10  add 1 to N                    N: 7
11  end-perform
```

There is no complex recurrence relation to solve here; it is a simple matter to determine that the expression representing the number of iterations is:

a - 2

More complicated loop recurrence relations pose a more severe test for symbolic execution.

Ramamoorthy, Ho and Chen [Rama76] note that, in general, solving such recurrence relations is not a simple task. Cheatham [Chea79] reports that a technique for solving the more common recurrence relations has been developed. A symbolic expression to describe the number of iterations is produced.

3.9.2 Module calls

The term 'module call' is used here to refer to the invocation of any out-of-line code. This includes sub-programs that are compiled separately from the invoking program, internal sub-routines, procedures and functions. The dilemma concerning module calls is whether to treat them using the macro-expansion approach [Boye75], the lemma approach [Chea79, Hant76] or using an approach which regards each call as an I/O boundary.

For example, in a COBOL program, when an out-of-line PERFORM is encountered during the symbolic execution of a path the execution may proceed by executing into the performed procedure. This is macro-expansion. Each time the performed paragraph is invoked the symbolic execution will be repeated, starting anew at each invocation. The lemma approach would symbolically execute a module once, and then use the results produced each time the module is invoked. The adoption of an I/O boundary would cause new values to be assigned to the returned parameters.

A reasonable approach might be to employ macro expansion for internal subroutines where global variables are used, such as the COBOL performed paragraph, and to use the lemma

approach where only parameters are passed. When external routines are called the parameters passed to the invoked routine can be treated as output results and the returned values as input which are given new symbolic values. Each separately compiled program is thus evaluated independently. Parameters passed to and from invoked modules should be regarded as output and input variables respectively. This approach is in keeping with the aims of decomposition namely the division of problems into units of manageable proportions and cohesive functions.

3.9.3 Arrays

Arrays can be problematic for symbolic execution [Clar76, King76, Rama76, Howd77]. There is no difficulty for an expression such as:

move B to A(5)

because A(5) is unique in the same sense that B is unique. Both refer to a specific single element and this enables symbolic execution to proceed in the usual way. The difficulty arises when the subscript is an input variable, or is dependent upon at least one input variable, for example:

move B to A(I)

where I is an input variable. Symbolic execution cannot proceed in the usual way beyond this assignment, because the identification of the element within the array is not defined and is said to be an ambiguous array element.

One approach replaces array references with an n-way branching construct containing one branch for each array element. For example, where A is defined as an array with four

elements:

```
move B to A(I)
```

is replaced by

```
evaluate I
  1  move B to A(1)
  2  move B to A(2)
  3  move B to A(3)
  4  move B to A(4)
end-evaluate
```

This approach increases the number of paths substantially even for small arrays. Where the number of elements in an array is large this approach is impractical.

Alternatively, when the PC is to be used to derive a test case the ambiguity can persist during symbolic execution and be resolved at creation of the test case. This approach needs special processing because path infeasibility cannot be assessed in the usual way.

3.9.4 Identifying infeasible paths

Of the 33 paths through the triangle program in figures 3.1 and 3.2 only 6 are feasible. Symbolic execution must be capable of determining when constraints on a path are contradictory and hence the path is infeasible.

Clarke and Richardson [Clar81] describe two approaches to determining infeasibility: axiomatic and algebraic. The axiomatic technique makes use of a theorem proving system to determine whether the constraints are contradictory. This technique is not discussed further. The algebraic technique uses the simple conditions within the path condition as a set of constraints. An artificial objective function is created, for example, the sum of the variables

present in the predicates. If this optimization problem can be solved then the path is feasible. The solution may be used as a test case to execute the path.

When no solution can be found an indication as to which constraints contribute to the infeasibility would be useful. This is not easily obtained because the algorithms employed do not readily identify the constraint which contributes most to the infeasibility.

A more informative approach might be to submit the first and second constraints on the path to the optimizer for solution. If this proves to be feasible then add the next constraint on the path and attempt to solve. This addition of constraints is repeated until either the path is complete or the constraints are infeasible. This approach identifies the constraint which first causes infeasibility.

The reverse approach could also be adopted by attempting to solve for the complete set of path constraints. If the path proves to be infeasible then the last constraint is removed and solution attempted for the remaining constraints. Continue removing constraints until a feasible solution is determined.

A binary division approach could be employed. First, attempt to find a solution for the complete path. If infeasible try a set of constraints for the top (first) half of the path. Continue halving the unknown set of constraints until the offending constraint is determined. This approach, if useful, is likely to be appropriate only for paths with a large number of

predicates.

The most useful approach combines path selection and symbolic execution undertaking feasibility assessments after each branch selection is made. If the addition of the selected predicate causes infeasibility an alternative predicate is chosen and its feasibility reassessed.

CHAPTER FOUR APPLICATIONS OF SYMBOLIC EXECUTION

The uses for the output from symbolic execution may be classified into: path domain checking; test data generation; assertion checking; and program reduction. Software maintenance can also benefit from symbolic execution when placing changes in a program and in aiding regression testing.

4.1 Path domain checking

When a path is executed with a single case it may result in:

(i) incorrect output due to one or more faults (universally incorrect);

(ii) correct output although a fault exists (coincidentally correct);

(iii) correct output and no faults exist (universally correct).

Distinguishing between a coincidentally correct output and a universally correct output requires more than execution of a path by a single case. The output from a symbolic execution is more general, as it represents all the cases that could execute the path. Individual case values may be substituted into the symbolic expression and evaluated by hand but this is laborious.

It may be worthwhile for the tester to select cases from the test case domain such that both minimum and maximum values for the variables are used to evaluate the expression. For example, consider the following variable declarations:

```
grosspay        pic 9(4)
taxfree         pic 9(4)
percent         pic 9(3)
netpay          pic s9(4) sign leading separate.
```

Here grosspay, taxfree and percent are input variables and netpay is output. In addition to the picture declaration percent is constrained to the range 0 to 100. The program fragment introduced at the beginning of Chapter 3 is used to determine the value of netpay. Minimum and maximum values would be as follows:

```
Minimum         grosspay  = 0000
                percent   = 000     giving    netpay = 0000
                taxfree   = 0000

Maximum         grosspay  = 9999
                percent   = 100     giving    netpay = 9999
                taxfree   = 9999
```

Mixing minimum values for some variables with maximum values for other variables may also prove valuable eg:

```
                grosspay  = 0000
                percent   = 100     giving    netpay = -9999
                taxfree   = 9999
```

This result appears to suggest a need for further constraints in the program such as:

```
grosspay >= taxfree
```

to allow only the creation of positive netpay values.

4.2 Test data generation

The expression produced for each output variable when substituted with actual values in place of symbolic values effectively becomes a test case. During the process of path domain checking the task of selecting values in place of symbolic values is performed by hand. The tester's knowledge of possible problem cases (heuristics) may be used to create more stressful cases. Alternatively the process may be automated by using an optimizer.

The path condition is used as a set of constraints. The objective function may be any expression containing the variables present in the path condition. One possible strategy produces many test cases for each path. The symbolic expression for each output variable on a path is used in turn as an objective function. The optimizer is executed twice for each objective function, once to maximize it and once to minimize. This results in each path being executed twice as many times as there are output variables. For example, if a path outputs three variables then six cases would be generated.

Given that the functions are continuous this would be a useful set of cases increasing confidence in the likely outputs of all intermediate values.

Where the constraints are not easily solved techniques using random selection of values may be used [Rama76, Hedl81]. When using random selection there is a practical need to set a range of values from which to select. Failure to set such a range results in large number solutions which cause overflow on execution. Unfortunately, setting upper and lower bounds may remove a solution from consideration. If no solution is found with the range imposed then the bounds must be relaxed incrementally until a solution is discovered.

Hedley [Hedl81] discovered that the only predicates remaining unsolved due to an imposed range on the values were those which contained numeric constants. Through experimentation he set out to determine the closest bounds that would generally produce solutions. He found these to be plus and minus twice the largest absolute value found in the predicates.

4.2.1 Partition analysis

Partition analysis is a technique that makes use of the output from symbolic execution to determine test data [Rich81]. It uses symbolic execution to identify sub-domains of the input data domain. Symbolic execution is performed on both the program and the specification. The resulting expressions are used to produce the sub-domains such that each sub-domain is treated identically by both the program and the specification. Where a part of the input domain cannot be allocated to such a sub-domain then either a structural or functional fault has been discovered in either the program or the specification. In the system described the specification is expressed in a manner close to program code. For this technique to prove practical, enhancements are required such that specifications can be expressed in a less program-like format.

4.3 Assertion checking

Assertions can range from a general, and hence usually complex, statement about say the contents of a table, to a simple statement about the value of a single variable. Simple assertions can be verified by symbolic execution but more complex assertions cannot be easily accommodated.

A simple assertion may be placed on any branch in a program. It need not affect the normal execution of the program but can be used during symbolic execution to assess the validity of the path. When symbolic execution along a path encounters an assertion it is treated in the same way as any other predicate. If its truth can be resolved then symbolic execution either

halts due to infeasibility in the case of false, or is discarded in the case of true. If its truth is unresolved then it is conjoined on to the PC.

When the addition of assertions to a PC turns a feasible path into an infeasible path then the assertions have been violated. This indicates that an error is present in the path (or in the specification of the assertion).

The following straight-line form of a path contains assertions marked by 'a'. Suppose that an error is present indicated by 'e'.

```
 1          move 0 to total
          e move -1 to counter          move 0 to counter
 2        a [total >= 0]
          a [counter >= 0]
            accept score
            move "continue" to action
 3          [score < 1]
 6          [score <>0]
            move "stop" to action
15          [action = "stop"]
          a [average <= maxscore]
          a [average >= 0]
19          [counter > 0]
            display "No values have been input"
24          [action = "stop"]
```

Symbolic execution proceeds in the usual way. On reaching branch 2 the PC is empty. The assertion 'total >= 0' can instantly be verified as true because total has just been set to 0. The assertion need not therefore be placed on the PC. The assertion 'counter >= 0' can also be verified but in this case it is false. Counter has previously been set to -1 in direct conflict with the assertion. Symbolic execution fails due to the failed assertion which arose because of the incorrect initialization of counter.

Assume now that a path is undergoing symbolic execution and is reaching the end of branch 16. The current expression for average is represented by:

(score{1} + score{2} + score{3}) / 3

where score{n} represents the nth value input to score.

The assertion 'average <= maxscore' is now substituted with the expressions for average and maxscore giving:

(score{1} + score{2} + score{3}) / 3 <= 7

The truth of this predicate cannot be evaluated so it is placed on the PC. If there is a solution to the final PC then the assertion is upheld. If there is no solution to the final PC even though the PC without the assertion is feasible then the assertion fails. However, placing an assertion on a path that is infeasible determines nothing about the validity of the assertion.

4.4 Program reduction

King describes how symbolic execution can be used to achieve program reduction [King81]. This is the act of taking a program and producing another program containing fewer statements. The result is "a simpler program consistent with the original but, operating over a smaller domain" [King81]. This is useful when reusing software where only a subset of the cases handled are required. A major step forward will have taken place in software engineering when the reuse of software is normal practice. Program reduction is a step towards this goal. This technique is not considered further.

4.5 Software maintenance

Two problems facing the software maintainer are avoiding the introduction of unexecutable code and coping with the large volume of regression tests required.

In preparing program changes the maintainer often spends time on ascertaining the constraints which govern the variables at the proposed points of insertion of the new code. This effort is required to avoid the introduction of unexecutable code. The converse of this situation also arises. That is, unexecutable code is introduced because the maintainer is unaware of the prevailing constraints.

It is likely that many of the paths executed during regression testing are executed more than once in an attempt to overcome the risk of coincidental correctness camouflaging an error.

Symbolic execution can be used to reduce the likelihood of introducing unexecutable code, to speed up the process of determining the constraints on variables at a given point in a program and to reduce the volume of regression testing.

4.5.1 Modifications

When modifying software the impact of the changes can be categorised as either a domain change or a computation change. A domain change is caused by introducing a new selection statement or by changing an existing one. This creates new paths or changes the range of values that can execute some existing paths. Computation changes affect the outputs that

result from executing a path. Computation changes often cause a different path to be followed for a given input case but these can be regarded as side-effects rather than as primary changes. In many instances new computations are introduced for specific domains. Ensuring that the intended domain and only the intended domain is affected by a new computation is a major part of the maintenance activity.

The areas of concern when modifying software can be classified as follows:

1. Are the given domains processed by the new code ?
2. Are all other domains not processed by the new code ?
3. Has the introduction of a selection statement introduced a new domain or has an unnecessary selection statement been introduced ?

A Class 1 error occurs when only a subset of the intended domain is processed by the newly inserted code. Suppose some process, X, is to be undertaken for values of A in the range 10-19 inclusive. In the following program fragment lines 15-18 have been inserted with this intention.

```
01    A pic 99
      :
10    if A > 10
11    then
        :
15      if A < 20
16      then
17        do X
18      end-if
        :
31    end-if
```

At line 11 A can take a value in the range 11-99 inclusive. At line 15 the range of values of A is reduced to 11-19 inclusive. As a result, A = 10 is not included in the domain being subjected to process X.

A Class 2 error occurs when a superset of the intended domain is processed by the newly inserted code. Consider again the process X which is again to be undertaken for values of A in the range 10-19 inclusive. In the following different program fragment lines 15-18 have been inserted with this intention.

```
01    A pic 99
      :
10    if A > 9
11    then
        :
15      if A < 21
16      then
17        do X
18      end-if
        :
31    end-if
```

At line 11 A can take a value in the range 10-99 inclusive. At line 15 the range of values of A is reduced to 10-20 inclusive. As a result the domain A=10-19 is subjected to process X but, in addition, A=20 is also subjected to X.

Class 3 errors occur when a selection statement includes values that are excluded by previous selection statements thus creating branches that cannot be executed. For example consider the following program fragment:

```
10    if A > 10
11    then
        :
20      if A < 50
21      then
          :
25        if A > 75
26        then
            :
29        end-if
          :
31      end-if
32    end-if
```

At line 21 the range of values that A could take is 11-49. The introduction of lines 25, 26,.. is superfluous because A must be less than 50 to reach this point, the additional condition of A > 75 has already been guaranteed to be false.

Class 4 errors occur when the truth of the condition in a selection statement has already been established by earlier selection statements. Consider again the example used above when discussing class 3 errors, except that line 25 is changed to A < 75. The selection at line 20 has guaranteed that A < 50 is true, so the test of A < 75 is superfluous as it will always be true.

4.5.2 Impact of module hierarchy on path infeasibility

Avoiding infeasible paths during software maintenance is a difficulty compounded by the impact of modules higher up the calling hierarchy. These superordinate modules determine the domains of the input parameters and so are integral to the problem of infeasible paths. When a conditional statement is inserted such that no feasible path is found for its branches the conditional predicates that contribute to the infeasibility may be found either locally, in the calling module or, in a module higher up the calling hierarchy.

When faced with changing one module out of a large system of modules the maintainer faces an escalated form of the infeasible path problem. It is not sufficient to be able to place the modification on a path assessed as feasible in the module, if the constraints placed on that module by the calling modules are such that the new code cannot be executed. What is required is a means of assessing feasibility - both locally and for the wider chain of module invocations.

4.5.3 Regression testing

To avoid unintentional changes to existing parts of a program going unnoticed a rigorous testing phase is often undertaken. This is known as regression testing. Here a set of test cases are executed and the results compared to the results obtained from the same test cases run through the unmodified version of a system. If the new version has remained unaltered for these functions, the results for these cases from both versions of the program will match. Any differences indicate an unexpected change.

A large set of test cases may have been used to test the software when it was first developed. It is useful to identify a subset of these tests for regression testing. Work on selection of test cases for revalidation has been undertaken by: Fischer [Fisc77]; Yau and Kashimoto [Yau89]; and Hartman and Robson [Hart90].

Yau and Kashimoto's approach first constructs a cause-effect graph for the newly constructed program and then divides the program's input domain into several classes. These classes are

used to generate test cases. After the program has been modified the input partitions are derived again and the original set of test cases searched for a subset of cases which provide coverage of each of the new partitions. Those partitions which cannot be covered require the generation of new cases. Both these new cases and the sub-set of the original cases are then executed with parallel symbolic execution. Symbolic execution is undertaken to provide debugging information.

This approach is reported to be successful with programs that undertake classification of input, such as determining whether three integers represent either an equilateral or isosceles or a scalene triangle, but is less successful for algorithmic programs such as sorting.

Fischer proposes the use of a 0-1 integer programming technique to determine the test case to be used for regression testing. The program is parsed and represented as three matrices: connectivity; reachability; and variable set/use. The first two represent the controlflow of the program and the third contains details of the dataflow. A fourth matrix - test case dependency - records the coverage of each test case. The objective of the approach is to determine the minimum number of test cases to provide coverage of the program.

Hartmann and Robson are attempting to extend Fischer's method by including parameter information in the set/use matrix. They also intend to apply it to several languages and to develop tools to automate test case selection.

Fisher's approach, including the extensions proposed by Hartmann and Robson, presumes that the execution of a branch just once provides an adequate test. Unfortunately, adoption of this technique will not overcome the problem of coincidental correctness: a test case may yield the correct output for a case where another case causing execution of the same path would yield an erroneous result. To guard against the problem of coincidental correctness multiple path executions are necessary, but still not sufficient.

4.5.4 Avoiding infeasible modifications using symbolic execution

The practice of treating module calls as an I-O boundary in symbolic execution is not helpful when infeasiblity is caused by conflicting constraints which are in different modules. Adoption of the macro-expansion approach to symbolic execution overcomes this problem. This could suffer from the path explosion problem, where there are many paths available. However, judicious use of a path selection strategy, such as coverage of a branch at least once, avoids this problem, and symbolic execution provides a useful means of assessing all three maintenance problems which are caused by domain and computation errors.

Errors where either a sub-set or super-set of the correct domain is processed are easily detected. Test cases can be generated at the minimum and maximum values for the variables in the newly inserted conditional statements. If these do not cater for the extremes of the domains to be covered, or include values not intended, then a domain error is identified, even without executing the generated test case.

Situations where the introduction of a particular conditional statement would be superfluous can also be identified using symbolic execution. An assertion postulating the opposite of the intended condition is inserted at the point under consideration for the conditional statement, for example, in the program fragment describing class 3 modifications errors in Section 4.5.1, the assertion would be A <= 75. If this assertion is not upheld, then there is no need for the conditional statement on this path. However, the number of paths to be investigated before the conditional statement can confidently be dropped should not be underestimated.

4.5.5 Regression testing using symbolic execution

Rather than execute all the original test cases the set of paths covered by these tests should be established. The result is a set of critical regression testing paths. Each of these critical paths can then be symbolically executed (just once) for both the old and new version of the program.

For each path, the path condition is compared between the two versions and the output expressions for each variable are also compared. When the path conditions do not match a domain error has been detected. When the path conditions match but one or more of the variable expressions do not, then a computation error has been identified. This technique is capable of replacing many conventional test executions by just one symbolic execution.

CHAPTER FIVE EXISTING SYMBOLIC EXECUTION TESTING SYSTEMS

EXDAMS [Balz69] is a monitoring system constructed before the first symbolic execution systems. It requires the program being tested to be executed with data values. Whilst the program executes it is closely monitored and a history of the execution is stored. It provides a flexible approach to the examination of the execution history. An analysis of the source program is undertaken to provide a data table and model of the program. These are used in conjunction with the history to provide the user with a view of the program's state at any point during the execution.

After EXDAMS, many of the software testing and debugging tools made use of symbolic execution. Some of these systems incorporate features similar to those provided by EXDAMS. There are now thirteen systems described in the literature whose authors state that use is made of symbolic execution. These are EFFIGY [King75, King76], SELECT [Boye75], ATTEST [Clar76a, Clar76b], CASEGEN [Rama76], DISSECT [Howd77, Howd78], EL1 Symbolic Evaluator [Cheat79, Ploe79], SMOTL [Bice79], Interactive Programming System [Asir79], SADAT [Voge80], the FORTRAN Test-bed [Hedl81], IVTS [Tayl83], UNISEX [Kemm85] and SYMBAD [Coen90].

Additionally, MALPAS [Webb87, O'Ne88] and SPADE [Carr86, O'Ne89] are two commercially available systems for which there are no detailed descriptions in the academic literature. The few papers that do describe the systems outline the facilities provided but give no description of their inner workings. However, both MALPAS and SPADE appear to make

use of symbolic execution.

The nature of the symbolic execution utilized in the fifteen systems varies from one system to the next. Some of the approaches may not even constitute symbolic execution.

5.1 Minimum features of a symbolic execution testing system

The minimum features that a system should exhibit before it can be considered to make use of symbolic execution may be as follows:

* it produces a path condition for each path examined;
* it determines whether a path condition is feasible;
* for each output variable it produces an expression in terms of input variables and constants.

Using these three features as criteria for rejection causes EXDAMS, DISSECT, EL1 symbolic executor, SMOTL, SADAT, IVTS and UNISEX to be removed from the list of symbolic execution systems.

EXDAMS has never been claimed as a symbolic execution system. Paths are executed with actual data and no symbolic expressions are maintained. Hence, no feasibility checking can be undertaken.

The DISSECT system produces expressions for variables and a PC for each path specified.

It fails to check the consistency of the predicates of the PC; this task is left for the user to undertake by hand.

The EL1 symbolic evaluator does not create symbolic expressions for output variables nor does it determine path feasibility.

SMOTL does not create expressions for variables nor maintain a PC. Its path analysis is based on maintaining maximum and minimum values for each variable. It uses these values to check for predicate contradiction and hence path feasibility. This is not symbolic execution but, nevertheless, it achieves one of the results that can be produced by symbolic execution - the generation of test cases for a path. Other results achieved using symbolic execution, such as assertion checking, are derived from the expressions maintained for the path condition and each variable. SMOTL cannot achieve these results because the expressions are not maintained by the system. However, SMOTL does have a strategy for combining paths in an attempt to reduce the number required to cover all branches. This technique may be applicable in all path based testing systems, including those using symbolic execution.

SADAT does not attempt to determine path feasibility: it leaves this for the user to perform by hand. Expressions are not determined for output variables.

IVTS is still under development and it is the symbolic executor which is incomplete. As yet, the system does not determine path feasibility.

UNISEX is intended to use an expression simplifier and a theorem prover to assess path feasibility. In some cases the expression simplifier can resolve a PC to either true or false. When this cannot be achieved the resulting expression will be passed to the theorem prover. The theorem prover has not yet been built. Currently the PC is output for the user to assess by hand.

MALPAS and SPADE are two commercially available validation systems which both stem from work sponsored by the Ministry of Defence. The systems appear to share some features which is a result of their common origin. There are a few papers in the literature describing these systems [Carr86, Webb87, O'Ne88, O'Ne89] but there are no published papers describing their inner workings and the problems encountered in their development. In addition to the summary papers publicity material is readily available. The following descriptions are based on both of these sources.

MALPAS (MALvern Program Analysis Suite) transforms the source program into intermediate form and determines the number of paths through the program. This is straightforward in the simple example shown but it is not clear how this would work for a more practical program containing loops and an infinite number of paths.

A static analysis is undertaken to spot anomalies such as two writes without an intervening assignment. All the input variables which determine the value of each output variable on each path are identified together with a list of the predicates at each node which must hold for the

path to be executed.

Finally, the system determines which of the paths are feasible and which infeasible. No description is given of how this is achieved but, given the data that is maintained to this point, it looks likely that some form of symbolic execution is undertaken together with feasibility checking of the resultant path condition. Results are displayed for each path in the following form:

```
if path condition
then
  var-1 = expression-1
  :
  var-n = expression-n
end-if
```

Assertions may be inserted into the source program which are checked against the path condition. This cannot always be achieved; in such a case the expression is output for the programmer to inspect. It is not stated how this assessment is made nor under what circumstances the assessment cannot be made.

SPADE (Southampton Program Analysis and Development Environment) produces an intermediate form known as the functional description language (FDL). At present translators exist for Pascal and INTEL-8080, and translators for Ada and Modula-2 are planned. The functional description language reader assesses the intermediate form for syntactic errors. This seems to be unnecessary as this duplicates processing undertaken by the language compiler. However, users can write FDL directly as a specification and the FDL reader is then used as the only syntax checker. Anomalies such as unreachable code and unused variables are identified as well as an analysis of module interfaces.

It is not clear from the published material how paths are selected, but path conditions are established. Symbolic execution of a path is undertaken and the output is claimed to be 'useful' for test data selection but it does not generate specific test cases. No details are given about the assessment of path infeasibility.

The proof checker is used as a documentation tool for proving the program. It contains tools for arithmetic and logical expression simplification and limited automatic deduction facilities. A data base of rules is maintained. The user guides the checker in search of a proof.

By excluding the systems which do not exhibit the three crucial criteria and those systems for which there is little detailed material in the literature, there remain seven systems: EFFIGY, SELECT, ATTEST, CASEGEN, the interactive programming system, the FORTRAN test-bed and SYMBAD. These systems can be considered symbolic execution testing systems.

5.2 Strengths and weaknesses of the symbolic execution testing systems

The strengths that should be incorporated into future symbolic execution systems and the weaknesses and omissions that should be overcome are identified for each of the seven systems.

5.2.1 EFFIGY - An interactive symbolic execution system

EFFIGY was the first system to make use of symbolic execution. It was designed to allow a program to be developed and tested gradually, making use of the symbolic execution facility

as well as execution with actual data values. It is a development tool which can be used when testing either program fragments as they are written, or complete programs.

The interactive nature of the system is one of its strengths since it allows a degree of flexibility. When a path is being evaluated the user may insert actual values as well as symbolic values. This results in expressions for some variables containing a mixture of symbolic and actual values. A symbolic trace of a path can be obtained. This shows the state of specified variables line by line.

A powerful feature of EFFIGY is the use of assert statements which can be verified, based upon the symbolic representations of the variables and PC.

Perhaps the most important omission from the EFFIGY system, given that it is interactive, is the absence of a facility to provide a view of the flowgraph or a coverage metric. It is possible that a user could terminate a session with EFFIGY having discovered many faults, corrected them, and been satisfied with the modifications but have omitted some branches of the program. It would be helpful if an indication of the extent to which the analysis has covered the flowgraph was provided. No long term recording of coverage statistics appears to be maintained, though it is stated that an exhaustive 'test manager', of which no details are given, is a part of the system.

No strategy for path selection is incorporated into the system. The users must employ their

own testing heuristics to create a strategy. The selection of the appropriate branch, when this is not determined by the state of the PC, is left for the user, so too, is any decision on the number of iterations to be made at each loop. Determining the feasibility of paths is achieved using a theorem prover.

There is no direct reference in the literature to how EFFIGY handles module calls. However, Hantler and King [Hant76] describe the lemma approach. Both of them worked on the construction of EFFIGY and it is likely that this is the method used in EFFIGY. Perhaps EFFIGY's major weakness is that it handles only a small subset of the language being analyzed.

The main weaknesses of EFFIGY can be summarized as:

* only a subset of the language can be used;
* no strategy for path selection;
* no output of coverage metrics.

5.2.2 SELECT - Symbolic Execution Language to Enable Comprehensive Testing

SELECT [Boye75] was constructed at about the same time as the EFFIGY system. It attempts to provide similar facilities to EFFIGY and in addition creates values which can act as test cases. These values are generated as a by-product of evaluating path conditions. The system was built to process programs written in a subset of LISP. It is not clear at what class of application the system is aimed.

SELECT produces 3 categories of output for each processed path:

* test data;
* simplified symbolic expressions for each program variable on a path;
* statement of correctness of user supplied assertions.

SELECT also identifies some infeasible program paths.

The system aims to be automatic and is claimed to have a path selection strategy where the aim is coverage of every branch. Paths are created by commencing at the start of the program and adding one branch at a time until a halt is reached. Each time a branch is added to the PC, it is passed to an inequality solver to determine path feasibility. Both linear and non-linear inequality solvers are employed. The solver maximizes an arbitrary objective function and the solution to the path condition of the whole path is used as a test case.

SELECT requires that, for loops which may be executed a variable number of times, the user specifies the number of iterations to be included in the path. The system updates the PC and variable expressions for each iteration of the loop.

Subroutines are tested in isolation until deemed satisfactory. Whenever a subroutine is invoked, the set of path conditions for that routine may be incorporated into the path which invokes it. This adds to the combinatorial explosion of the number of possible paths. Boyer suggests that a better approach might be to make use of input and output assertions for each

subroutine [Boye75]. This would not increase the number of paths.

The ambiguous array reference problem, where arrays are indexed by input variables, is tackled by the introduction of virtual paths. This adds to the complexity of the flowgraph by adding a branch for each element of the array.

The weaknesses of SELECT can be summarized as:

* only a subset of LISP can be used;
* approach to ambiguous array elements increases number of paths;
* use of macro-expansion increases number of paths.

5.2.3 ATTEST - System to generate test data and symbolically execute programs

The prime objective of ATTEST is to generate test data for a path [Clar76a, Clar76b]. This is done using the conditions and assignments of the path to ensure that the data will force execution of that path. A secondary output, produced almost as a by-product, is a symbolic representation of the output variables of the path in terms of the input variables and constants. The system will sometimes inform the user if the path is infeasible and cannot therefore be executed because no such data set exists. The system cannot detect all infeasible paths. In particular, it cannot identify infeasibility where the system of constraints is non-linear. This is surprising as one would expect FORTRAN programs to contain non-linear predicates.

ATTEST's main strength is significant. It analyzes FORTRAN programs and, hence, was a

step towards a widely usable symbolic execution testing system which can be used in either automatic or interactive mode.

The difficult task of selecting the paths for analysis is not tackled by the system, but left entirely to the user. Paths for analysis can be specified in two ways: statically or interactively. The static specification requires the whole path to be specified at once, whilst the interactive mode allows the user to select one branch at a time as each conditional statement is reached.

When a constraint makes the path infeasible the user is informed and the analysis proceeds to the next path. When the end of a path is reached and it is feasible then the final solution obtained is a test case that will cause execution of the path.

The system attempts to discover where array indexes stray out of bounds using two temporary constraints. One specifies that the upper bound must be exceeded and the other that the lower bound must not be achieved. For example, consider an array A(5..10). Constraints of index<5 and index>10 are used. When either of these temporary constraints is conjoined on to the path condition the constraint should prove contradictory if the path will not allow the index to stray out of bounds.

When ambiguous array elements are encountered the symbolic execution halts.

No path selection is undertaken by the system. The user must input the paths which are to be

evaluated. When path constraints are checked for feasibility they are passed only to a linear inequality solver. However, the system analyses FORTRAN programs which might be expected to contain non-linear constraints. Without user intervention to check for linear constraints, path feasibility checking may be unreliable.

The processing of module calls uses macro-expansion. This does not cause a path explosion problem for the system because no recording of path coverage appears to be undertaken. It merely increases the length of the path. There is no facility for insertion and verification of assertions.

The weaknesses of ATTEST can be summarized as:

* cannot detect infeasibility of all paths;
* no path selection;
* symbolic execution halts on encountering ambiguous array elements;
* no recording of test coverage;
* no facility for assertion checking.

5.2.4 CASEGEN - Automated program test data generator

CASEGEN generates test data and, like ATTEST, analyses FORTRAN programs [Rama76]. The system operates entirely in batch mode and consists of four components:

* a FORTRAN source code processor;
* a path generator;
* a path constraint generator;
* a test data generator.

The FORTRAN source code processor analyses the source and generates a data base consisting of a flowgraph, symbol table and an internal representation of the source code. The path generator uses the database to produce a set of paths to cover all branches. The authors do not discuss the path selection strategy, except to state that loops are executed a fixed number of times. For each path the path constraint generator produces the path condition. Some of the paths generated may be infeasible. The test data generator aims to create values for the input variables that satisfy the set of inequalities for each path, hence creating a test case for each path. The sets of inequalities in the path condition are solved using linear programming, integer programming, mixed programming and non-linear programming techniques as appropriate. A procedure based on systematic trial and error and random number generation is also used. This use of several types of optimizer makes this system superior to ATTEST in terms of feasibility checking. Nevertheless, it is error-prone. The numbers (26,7,7) were generated for the sides of an isosceles triangle [Rama76].

The builders of CASEGEN acknowledge the difficult aspects of symbolic execution.

The ambiguous array reference is retained during symbolic execution and is resolved during test data generation. Whilst the builders acknowledge the difficulties of handling module calls, they do not make clear what approach has been adopted in CASEGEN. Output of the variable expressions and PC are not provided and there are no facilities for insertion and evaluation of assertions.

The weaknesses of CASEGEN can be summarized as:

* no output of path condition;
* no output of expressions for variables;
* does not process assertions.

5.2.5 Interactive programming system

The Interactive Programming System [Asir79] is a collection of integrated software support tools for the design, development and maintenance of large computer programs. It was built as a general software development tool for a language available on a small computer: MINIPL/1. It is a pity that a full version of a widely available language was not used. There is no statement in the literature concerning the success of the system in the field but it was built in conjunction with Olivetti and was destined for commercial use.

The interactive part of the system is concerned with the direction of the symbolic execution. Its strengths centre around the state recording and review facilities. Whenever a conditional statement is encountered the user is required to select the branch to be pursued. The state of

the PC and variables at each conditional statement are recorded. The user may return to any previously encountered conditional statement and examine the states of PC and variables.

Assertions may be introduced into the program. The system will determine the consistency between the assertion and the path condition. The flexibility provided by the interactive nature of the system is used to overcome the absence of a path selection strategy. Whilst arrays and calls are handled the mechanisms employed are not made clear.

The weaknesses of the Interactive Programming System can be summarized as:

* only a small subset of input language;
* no path selection;
* not published how arrays and modules are handled.

5.2.6 FORTRAN test-bed

The test-bed [Hedl81, Henn83] undertakes static and dynamic analysis of FORTRAN programs and the construction of paths. It also generates test cases which will cause the execution of the paths. It is the path construction and test case generation that are of concern here as these make use of symbolic execution.

The test-bed makes use of the LCSAJ [Henn76]. An LCSAJ (linear code sequence and jump) is a series of statements ending with a transfer of control out of the linear code sequence. Paths can be viewed as a series of LCSAJs. Determining whether each LCSAJ is feasible can

be the first step in determining feasibility of a path. Should a single LCSAJ be infeasible then any path of which it is a component is infeasible. A path whose LCSAJs are all feasible, is not necessarily feasible. Two LCSAJs together may give rise to infeasibility. Gradually LCSAJs may be added together until either the addition of an LCSAJ results in the creation of an infeasible path or the path is complete and feasible. The path selection strategy does not deal with loop iterations but leaves this for the user to specify.

The FORTRAN test-bed determines the feasibility of an LCSAJ or a series of LCSAJs by employing symbolic execution. The system will solve sets of linear inequalities, provided by the path condition, to produce test data. The system makes use of random number generation to solve non-linear inequalities. This approach is similar to that employed by CASEGEN. The FORTRAN test-bed has an additional feature which makes use of a function to establish bounds for the random number generation. It is based on the constants in the inequalities. In a series of tests the system solved and generated test data for paths that covered all but 2.7% of LCSAJs.

Most module calls are handled by macro-expansion. This increases the number of paths available for consideration. Functions, however, are treated as input statements and new symbolic values are provided. It is not clear why different strategies are employed for function calls and other calls. Treating both as I/O interfaces would maintain the boundary identified during design.

The test-bed does not cater for the inclusion of assertions.

The system analyses FORTRAN programs and, with a few minor exceptions, this is ANSI standard FORTRAN. Symbolic execution is used as one technique employed by a software testing system that provides a variety of testing facilities such as static analysis through to test case generation. The test-bed is a commercially used system and cannot be regarded as just a researcher's experimental system.

The weaknesses of the FORTRAN-testbed can be summarized as:

* does not process assertions.

5.2.7 SYMBAD - SYMBolic executor of sequential ADa programs

This is the most recent system to emerge, development having taken place in parallel with the work on SYM-BOL, described in part two. SYMBAD [Coen90] processes Ada units with the exception of tasks, the mechanism for defining parallel processes.

The first stage is translation into an intermediate form which is common LISP rather than a SYMBAD-specific form. The next stage is a combined path selection and symbolic execution. When a branch point in the program is encountered the current path condition is examined by a theorem prover to determine whether the encountered branch condition is true, false or undetermined. When true or false symbolic execution continues down the relevant branch. When undetermined, the user selects the branch to be followed and the appropriate condition

is conjoined to the path condition. Symbolic execution continues down the chosen branch.

Assertions are checked by the system. They are embedded within the source code as comments and are checked against the path condition when they are encountered during the symbolic execution.

The authors claim to have incorporated a new method of processing arrays which overcomes the difficulty of ambiguous array references. An array is represented by an ordered set of pairs, where each pair contains details of an assignment encountered on the path. The first item in the pair is the value of the index, the second is the value assigned to that element. Consider the variable 'A' declared as follows:

```
A : array (1..max) of integer
```

and the following program fragment:

```
 8    get(J);
 9    A(J):= 5;
10    A(J+1):= 7;
11    if A(4) > 1
      :
      :
```

When variable 'A' is declared it is given the pair:

```
A : (any, undef)
```

indicating that all elements of the array are undefined. At line 8 'J' is arbitrarily assigned a symbolic value:

```
J : J@1
```

At line 9 'A' has a second pair added to its set:

```
A : (any, undef) (J@1, 5)
```

and after line 10:

```
A : (any, undef) (J@1, 5) (J@1+1, 7)
```

Line 11 is unresolved because the value of `J@1` is symbolic so the user must choose a branch, say the true branch. The predicate must now be placed on the path condition. By searching the set of pairs belonging to 'A' it can be seen that one of the following must be true for A(4) > 1 to be true:

```
J@1      = 4
J@1 + 1 = 4
```

The PC cannot sensibly contain both of these predicates as they are in direct contradiction so one must be chosen, but which one. This is not discussed by the authors, which is unfortunate because this is the heart of the array problem. Eventually, a choice must be made and this means selecting an actual value.

This method appears to be similar to earlier methods which deferred resolution of ambiguous array references until test data generation. In those systems paths were not assessed for feasibility so this posed no problem. Here, path feasibility is assessed, so a decision must be made. The method does have the benefit of maintaining information until a branch point, which requires the resolution of the ambiguity, is reached. However, experience suggests that such branches follow closely after the occurrence of the ambiguous array reference and so there may be only a small benefit to be gained from maintaining the ordered set of pairs.

No description is given of how module calls are processed. The system does not generate test cases. The path condition is output and the user must solve this by hand to create an appropriate test case for the path.

The weaknesses of SYMBAD can be summarized as:

* no test cases generated.

5.3 The ideal symbolic execution testing system

The notion of an ideal system ignores the practical considerations of construction. A system that delivers the features of an ideal system but which, for example, takes a long time to process, is clearly not ideal. Requirements of an ideal system are likely to be contradictory and a practical system must be a compromise between the conflicting requirements. Nevertheless, it is useful to set out an optimist's system against which existing and potential systems may be compared.

5.3.1 Input

The primary input to such a system is the source program which may contain assertions. Further inputs will be required in response to the results obtained:

* a specification of paths to be evaluated;
* changes to assertions;
* a simple mechanism for the input of the expected results for generated test data either as a range of values or a single value.

5.3.2 Output

The power of a symbolic execution testing system is determined by the variety and utility of the information provided. This should include:

* diagrammatic representation of the program flow-graph;
* paths (list of constituent statements);
* PC and an indication of feasibility or the constraint which turns the path infeasible;
* expressions for output and intermediate variables;
* statements on truth of each assertion;
* test data;
* results of execution;
* comparison of results v expectation;
* statement on coverage obtained in the testing already undertaken.

The utility of the output may be enhanced by providing a variety of viewing modes:

* a display of specified variables or PC on specified paths at specified points;
* a trace through the symbolic execution of a path at various speeds;
* a trace through the execution of user provided test cases;
* a trace through both symbolic and test case execution simultaneously;
* request a coverage report at any stage of testing.

5.3.3 Path selection

Current methods of path selection employ only simple strategies such as: take the left branch first, generate the shortest path. These strategies have a common target, that of achieving a

particular coverage metric such as all statements or all branches or all LCSAJs are executed at least once. Each of these strategies is prey to the problem of selecting infeasible paths. Having identified a set of paths which cover say, all branches, some of the selected paths will be found to be infeasible. This leaves the problem of identifying feasible paths which include the noncovered branches from the infeasible paths. This strategy does not consider the effectiveness of the paths selected and whether the paths are in some way more useful or interesting than other paths.

An alternative approach may be to attempt to identify paths and associated test cases that are, in some way, representative of a large set of cases. The ideal symbolic execution testing system should incorporate a path selection strategy in which the expressions produced by the symbolic execution are utilised in an attempt to identify 'interesting' paths.

The ideal system also requires novel solutions to the problems presented by loops, arrays and module calls.

5.3.4 A multi-language symbolic execution system

Researchers have reflected on the possibility of producing a general symbolic execution testing system which may be used regardless of the language in which the source program is written. Such a system should exhibit two necessary features. First, there is a need for a translator from each source language into the single intermediate representation processed by the evaluator. The intermediate representation must, therefore, cater for all features of the set

of source languages. Second, output messages from the execution system should be meaningful to the user. For example, when path infeasibility has been detected by the system the output should clearly identify the path and the predicate which turned the PC infeasible. The most instructive format for this output is where it refers to the source program submitted to the system. To achieve this requires the maintenance of references to the original source program for look-up before the output of messages.

An alternative attainable target may be a symbolic execution testing system that analyses programs written in a widely used language. For example, no symbolic execution system has been built for a commercial data processing language such as COBOL. Whatever the language, it should also handle the main features of that language. Adherence to a widely adopted standard, such as an ANSI standard, would be beneficial.

5.4 Ideal, existing and new symbolic execution systems

Tables 5.1, 5.2 and 5.3 summarize the features of the ideal and the seven systems described above.

The first system to be built that satisfied the minimum criteria for a symbolic execution system was EFFIGY. It is also EFFIGY that appears closest to the ideal system. All subsequent systems were, in some senses, a step backwards from the standard set by EFFIGY.

System	Author	Date	Language built in	Language analyzed	Input	Output
Ideal system			Widely available	Many widely available languages	Source program, assertions	Path, PC, Point becomes infeas Output variable expressions, Truth of assertions, Coverage details, Test data, Active/idle domains
EFFIGY	King	1975	not stated	Simple PL/1	Assertions, Display locations	Symbolic values Truth of assertions
SELECT	Boyer	1975	LISP	subset LISP	Source program, Assertions	Not stated
CASEGEN	Ramamoorthy	1976	FORTRAN	FORTRAN	Source program	Path, Test cases
ATTEST	Clarke	1976	FORTRAN	FORTRAN	Paths	Symbolic expressions, Test cases
IPS	Asirelli	1979	PL/1	Subset PL/1	Source program, Assertions, Branch selections	Symbolic expressions, Path cond State of assertions
FORTRAN Test-bed	Hedley	1981	ALGOL 68 FORTRAN	FORTRAN	Source program	Paths, Test cases
SYMBAD	Coen-Parisi	1990	C and LISP	sequential ada units	Source program, Assertions	Symbolic expressions, Path cond State of assertions

Table 5.1 Existing symbolic execution systems compared

System	Automatic/ Interactive	Path selection	Path feasibility	Assertion testing
Ideal system	Automatic and interactive	Uses symbolic expressions to aid in selection	Linear and non-linear, Optional: every branch, user request, or whole path	Yes
EFFIGY	Interactive	User selected	Every branch Theorem prover	Yes
SELECT	Automatic	All paths!	Every branch Algebraic linear and non-linear	Yes
CASEGEN	Automatic	Minimal set cover all branches	Every branch Algebraic linear non-linear random	No
ATTEST	Automatic and interactive	None	Every branch Algebraic only linear	No
IPS	Interactive	User control	When requested by user, Theorem prover	Yes
FORTRAN test-bed	Automatic	Not stated	Every branch Algebraic linear and random	No
SYMBAD	Interactive	User selected	Every branch linear theorem prover	Yes

Table 5.2 Existing symbolic execution systems compared

System	Call Handling	Loop Handling	Array Handling	File Handling	String Handling
Ideal system	User choice between macro-expan and I/O boundary	0,1,2 iterations plus minimum to give branch coverage plus user override	Maintain symbolic expressions with no path explosion	Maintain symbolic expressions by file output files of test cases and expected results	Maintain symbolic exprns for each charctr, assess feasblty string constants
EFFIGY	I/O boundary	User selects	1-dim	No	No
SELECT	Macro-expan	User specify max no of iterations	Yes	No	No
CASEGEN	Not stated	Fixed number	Yes	No	No
ATTEST	Macro-expan	Fixed max no of iterations	only constant indexes	No	No
IPS	Not stated	User selects	Not known	No	No
FORTRAN test-bed	Macro-expan, functions as I/O boundary	User specifies number of iterations	Maintains symbolic expressions	No	No
SYMBAD	Not stated	User selects	Yes	No	No

Table 5.3 Existing symbolic execution systems compared

The only major weaknesses of EFFIGY when compared to later systems are its failure to adopt a widespread language and its inability to devise test cases to satisfy the path condition. The authors do not state the language in which it is written but more important is the fact that it handles only a simple subset of PL/1. If the other systems were assessed in terms of their ability to handle similarly simple language subsets there is a possibility that they may appear at least as impressive as EFFIGY.

In a bid to make them more useful many of the subsequent systems handle larger language subsets and more popular languages. This has been achieved at the expense of a reduction in the capabilities of the systems. The two systems that are closest to EFFIGY in capability yet handle an almost complete and popular language are the FORTRAN test-bed and SYMBAD. If the FORTRAN testbed were to incorporate a mechanism for the specification of assertions then this would become closer to the ideal system than EFFIGY. SYMBAD and EFFIGY both require a means of generating test cases to satisfy the path condition to improve upon the FORTRAN testbed.

Assertions could of course be written as normal source code. This would mean that following testing the program would be in need of modification to remove them. It could be argued that permanent assertions with appropriate error messages could provide a powerful run-time semantic monitor.

The notion of the 'best' symbolic execution system is a useful one. It may be used as the

benchmark when assessing new systems. Unfortunately, it is difficult to determine which is the most useful system as there is no clearly outstanding system. EFFIGY perhaps comes the closest but a few additional requirements should be added to provide a useful benchmark.

If a new system provides the features provided by EFFIGY and processes a commonly used language and generates test cases then it would be a significant advance. The minimum features necessary to be comparable with this 'EFFIGY-plus' system are:

* input of source code containing assertions;
* interactive selection of paths;
* output of (a) symbolic values,
  (b) truth of assertions,
  (c) statement on path feasibility,
  (d) test cases.

Additional features necessary to justify the creation of a new system:

* widely used language for a class of programs not currently accommodated;
* path selection strategy for automatic systems.

It is the development of a path selection strategy that is fundamental to the advancement of all forms of path-based testing including the use of symbolic execution. Any tool that helps the user in selecting paths or includes an improved strategy for automatic path selection will be a contribution to research in path-based testing.

CHAPTER SIX RESEARCH AGENDA

This chapter has two purposes. First, it summarizes the weaknesses described in the earlier chapters concerning the use of symbolic execution as a testing technique and the weaknesses of the tools that have been constructed to support its application. Second, it establishes a set of research aims which will be addressed in part two of the thesis.

6.1 Weaknesses in existing research

The weaknesses of existing research may be divided into two categories. The first category contains the general problems identified for symbolic execution. The second category concerns the weaknesses of the tools built to support symbolic execution.

6.1.1 General problems of symbolic execution

There are four problem areas which are well documented in the literature concerning the application of symbolic execution. These are: path selection and the evaluation of loops; a dilemma over how to process module calls; the evaluation of array references dependent on input values and the checking of path feasibility. There are three further problems concerning the application of numerical optimizers for checking feasibility which are not documented in the literature. These are: formulating predicates of the form a<>b as a constraint; passing strings to an optimizer; and recognizing conflicting constraints when constraints contain both records and fields within records.

Current methods of path selection employ only simple strategies such as take the true branch first or generate the shortest path. These strategies have a common target, that of achieving

a particular coverage metric such as all statements or all branches are executed at least once. Each of these strategies is prey to the problem of selecting infeasible paths. Having identified a set of paths which cover say, all branches, some of the selected paths will be found to be infeasible. This leaves the problem of identifying feasible paths which include the noncovered branches from the infeasible paths.

Some systems that use symbolic execution have two distinct stages. First, select a path. Second, symbolically execute the selected path. Ideally, a symbolic execution testing system should incorporate a path selection strategy in which the use of path selection and symbolic execution is coordinated. Clarke, in the summary of a review paper [Clar85], suggests that "path selection and symbolic execution have a symbiotic relationship. Symbolic execution is used to guide the selection of paths, which are then symbolically executed. Thus, adaptive systems, where path selection and symbolic execution dynamically interact, should be considered". Clarke goes on to conclude "for the most part, current research is addressing the issues of verification, path selection, test data selection, debugging, optimization and development as independent topics. It is clear, however, that these topics are closely related and eventually should be integrated into a software development environment".

When considering modules the literature overemphasises the dilemma of choice between macro-expansion and the lemma approach. Macro-expansion increases the size of the program being examined but provides a more embracing coverage of the software. When maintaining a software system macro-expansion is a particularly useful feature as it allows the impact of

a change to be assessed, not just locally, but also throughout the system. When developing a new system the emphasis for symbolic execution may well be on the testing of small well bounded test units. The choice should be seen as a benefit to be valued rather than as a problem to be overcome.

Arrays are problematic because a static technique, symbolic execution, requires information, array subscripts, which is often determined dynamically at run-time. Current research sees this as an almost insurmountable difficulty. One solution proposes to introduce an n-way branch when an ambiguous array reference is encountered, where n is the number of elements in the array. This has the effect of increasing the number of branches significantly and, when branch coverage is being pursued, dramatically increases the number of test cases required such that at least one test case is required for each array element.

Symbolic execution must be capable of determining when constraints on a path are contradictory and hence the path is infeasible. This is usually reported as a problem for symbolic execution but it may more accurately be regarded as a feature of path selection. Further, it may be argued that symbolic execution is a part of a technique of assessing path feasibility rather than path infeasibility posing a problem for symbolic execution.

A common approach to assessing path feasibility is to formulate the path condition into an optimization problem. This requires that predicates from the path condition are formulated as constraints suitable for input to optimization software. Predicates of the following forms: a=b;

a<b; a<=b; a>b and a>=b are all easily formulated into suitable constraints. However, formulation is not straightforward for predicates of the form a<>b. This predicate cannot be passed to an optimizer because of the compound nature of the constraint. Both of the simple constraints a<b and a>b cannot be included in the same system of constraints because they are mutually exclusive and hence, together, they are infeasible.

A second problem facing constraint formulation is the occurrence of strings within predicates. A string cannot be passed to an optimizer because numeric inputs are required.

The occurrence of records in predicates also poses a problem for feasibility checking. Suppose one constraint stipulates equality of two records and that a second constraint stipulates the inequality of two variables. When the two variables are fields within the records present in the first constraint there is a contradiction between these constraints. Unfortunately, the optimizer will not be able to recognize the contradiction unless the relationship between record and fields is made explicit.

The latter three problems outlined above are not described in the literature and are addressed in more detail in Part Two.

6.1.2 Weaknesses of specific symbolic execution tools

One of the most glaring weaknesses in the set of existing symbolic execution tools is the absence of a system for commercial data processing languages such as COBOL. Only

Howden has referred to the symbolic execution of COBOL programs in his evaluation of several testing techniques [Howd78b]. His experiment undertook the symbolic execution of a COBOL program by hand. No researcher has given detailed consideration as to whether this class of software is suitable for symbolic execution or identified particular problems it may possess which would need to be overcome for its successful application.

As a result of this omission there are several features typical of this software class which are not considered in the literature:

* file handling;
* records in infeasibility checking;
* strings in infeasibility checking;
* high-level string processing constructs.

The literature contains descriptions of thirteen systems that claim to employ symbolic execution. The greatest weakness that these systems exhibit is that six of them do not provide **all** of the following basic features of a symbolic execution testing system:

* maintain for each variable an expression in terms of input variables and constants;
* produce a path condition for each path examined;
* determine whether a path condition is feasible.

These three features are fundamental to any tool that is intended to facilitate the application of symbolic execution.

The seven systems that include all three of the basic features have other weaknesses and

omissions but there are none common to all seven systems. Three of the six systems, ATTEST, CASEGEN and the FORTRAN test-bed, do not deal with assertions. It may well be coincidence, but all three are systems for analyzing FORTRAN programs. EFFIGY, SELECT and the Interactive Programming System process only a small sub-set of the target language. Four systems have no path selection strategy. ATTEST requires that paths be input to the system. This is a tedious task. EFFIGY, the Interactive Programming System and SYMBAD cater only for user selection of paths. Both EFFIGY and ATTEST fail to provide any indication of coverage which is a significant weakness in EFFIGY as the path selection is undertaken interactively. Ambiguous array references are problematic for two systems. ATTEST simply halts when they are encountered. SELECT generates a branch point with as many exit points as there are elements in the array; thus increasing the number of branches to be covered. SYMBAD maintains a history of the ambiguous array reference. It is not clear how the mechanism handles a branch-point that includes such a reference. CASEGEN maintains the ambiguity until the test data generation when the causal input variables are given values.

Other weaknesses apply to individual systems. ATTEST detects infeasibility only for systems of linear predicates. Detection of infeasibility in non-linear systems is left to the user. This is surprising for a FORTRAN analyzer as one would expect the software to contain non-linear predicates. SELECT is claimed to be an automatic system yet the user must supply the number of times that loops are to be iterated. CASEGEN does not output the expressions for variables and the path condition. This is unfortunate as they may be useful when debugging.

6.2 Research aims

The primary aim of the research is to demonstrate, or otherwise, that symbolic execution can be usefully applied to commercial data processing software. COBOL is considered representative of this class. The creation of a tool to support the application of a technique is a powerful demonstration that the technique can be applied. The aim is to produce a prototype COBOL symbolic execution testing system. The creation of a full system requires many person-years effort and is not a realistic target.

For a COBOL program the system should provide the following features:

* maintain for each variable an expression in terms of input variables and constants;
* produce a path condition for each path examined;
* determine whether a path condition is feasible;
* incorporate a strategy for automatic path selection;
* allow and facilitate user path selection;
* provide coverage metrics during user path selection;
* cater for ambiguous array references;
* assess path feasibility as each predicate is conjoined to the path condition;
* use the expressions generated by symbolic execution to reduce the selection of infeasible paths;
* verify assertions placed in the source program;
* allow user choice on: macro-expansion versus treating a module call as an input/output boundary.

The research aims can be summarized as:

1. Identify the problems facing the application of symbolic execution to commercial data processing software in particular to COBOL.
2. Propose means of overcoming the problems in creating a COBOL symbolic execution testing system.
3. Devise an approach to path selection that:
   a. selects more useful paths than existing symbolic execution systems;
   b. utilises the results of symbolic execution in a bid to reduce the likelihood of selecting infeasible paths.
4. Identify problems facing the use of a linear programming routine to assess the feasibility of paths and to overcome these problems demonstrating the practicality of the technique in a COBOL system.
5. Demonstrate that these proposals are practicable by constructing a prototype symbolic execution testing system for COBOL. It is not considered possible within the time-scale of a Ph.D. to build a full system.
6. Evaluate the symbolic execution testing system for COBOL.
7. Identify further work necessary to turn the prototype into a full working system and to identify areas in need of further research.

PART TWO

NEW WORK

CHAPTER SEVEN SYM-BOL: A SYMBOLIC EXECUTION TESTING SYSTEM FOR COBOL

This chapter provides an overview of the facilities provided by the SYM-BOL system. Chapters eight through eleven describe particular components of the system. Each of these chapters describes the problems presented by COBOL and the means used to overcome them. Some of the potential features that could be introduced in later versions of the system are described in chapter 12.

7.1 Environment

The SYM-BOL system is written almost exclusively in COBOL85 consisting of approximately 15,000 lines of source code. Some of the string processing might be more easily implemented in a language such as LISP. However, the bulk of the code in the system is concerned with housekeeping activities for which COBOL is more than adequate. Further, recent additions to COBOL in the 1985 standard have enhanced its string processing capabilities as well as generally bringing the language more up-to-date.

The system runs on a Microvax 3800 under VMS 5.3. The system is stand-alone with the exception of testing for path feasibility which uses the NAG-library routine E04MBF, which is a linear optimizer, to solve systems of linear constraints provided by the path condition. The feasibility checker is written in FORTRAN. Figure 7.1 shows the system architecture.

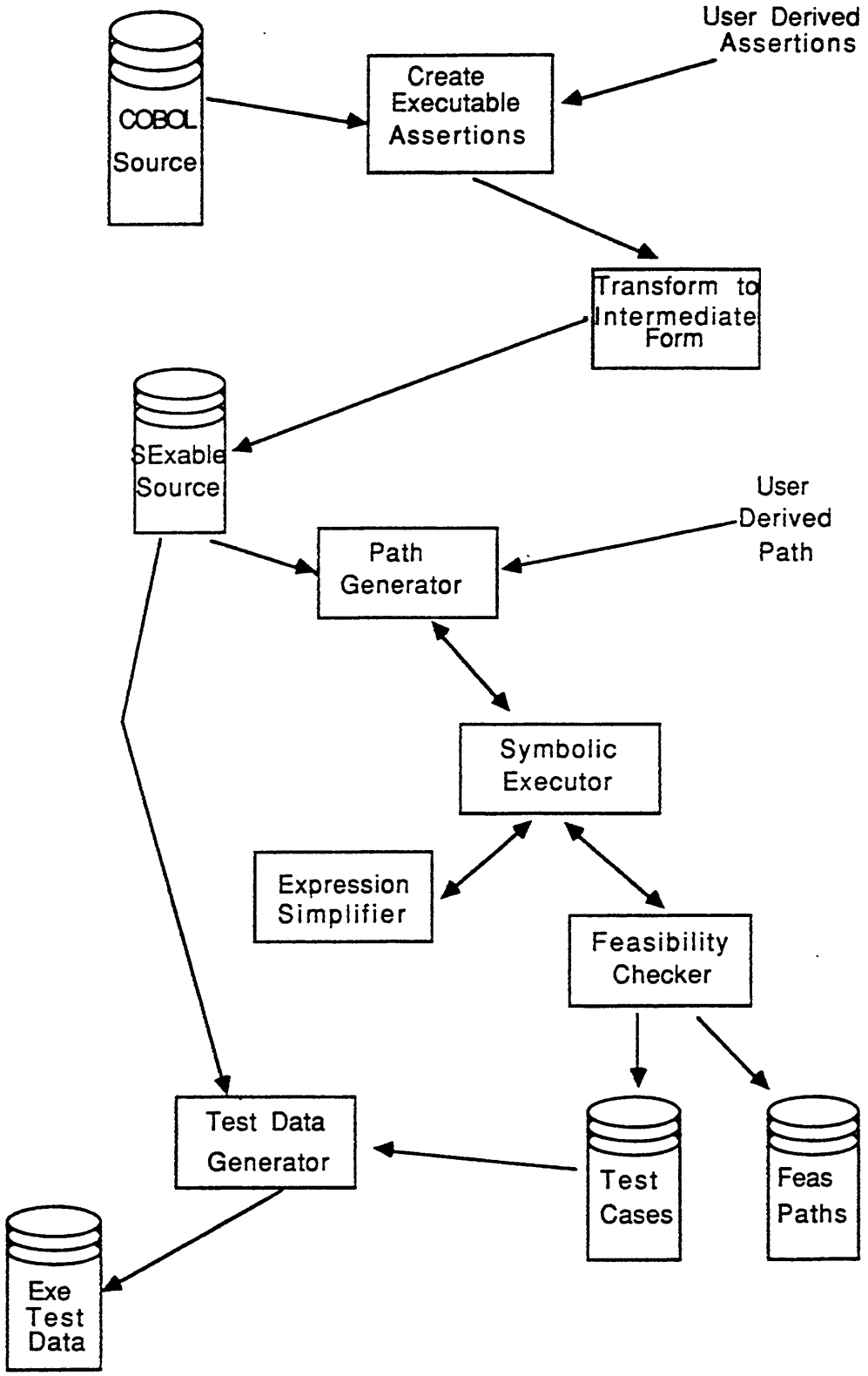

Figure 7.1 Architecture of the SYM-BOL system

7.2 Input

A large sub-set of standard COBOL can be used. The main restrictions are against indexed files, the string manipulation verbs and reference modification. However, a strategy for dealing with string manipulation has been developed and is to be included in a later version of the system. This strategy is described in chapter 12. A detailed list of the restrictions placed on the input source program is given in Appendix B.

The primary input to the system is the source program which may contain assertions. Further inputs will be required in response to the results obtained such as changes to assertions and selection of paths to be evaluated.

7.3 Output

* The paths which have been generated;
* Path Conditions plus indication of feasibility including truth of assertions;
* Symbolic expressions for variables;
* Files of test cases ready for execution;
* An indication of branch coverage.

7.4 User strategies

It is intended that the SYM-BOL system can be used in a variety of ways. First, it can select a set of paths and generate test cases to cause their execution. Second, it can generate test cases for paths supplied to the system. Third, it can be used as a debugging tool to help locate

errors detected by testing. Fourth, it can be used to search out a feasible path for a branch not covered by the testing undertaken so far. Some of these capabilities are not fully implemented, where this is the case it is stated in the following descriptions.

7.4.1 Path selection and test data generation

Path selection and test data generation are the main uses of the SYM-BOL system. It can be used to generate paths automatically or, perhaps more usefully, to assist the tester in selecting paths to be tested. Whichever mode of path selection is adopted symbolic execution is undertaken in parallel with the path selection. This allows the results of the symbolic execution to be used to help select the next branch for the path.

In the early versions of SYM-BOL feasibility checking was undertaken only once when the path was complete. This was because the NAG library and associated feasibility checking routines were developed on a Prime 9955 whilst the rest of the system was developed on a micro-VAX. The feasibility checker has been transferred on to the micro-VAX and the user may select the frequency of feasibility checking. It may now take place every time a predicate is conjoined to the path condition or just on completion of the path. There is no practical difficulty in undertaking frequent feasibility checking as most feasibility assessment appears to be determined after only one or two iterations of the optimizer. Further work is required to establish the generality of this result.

With immediate detection of infeasibility on branch selection the system forces the user or

the automatic path selector instantly to make an alternative branch selection. On completion of a feasible path the solution is retained as a test case. The case is stored and may later be transformed into records in the appropriate files ready for test execution of the program.

7.4.2 Test case generation for user supplied paths

Where a particular path has been identified as being of interest, the SYM-BOL system can be used to assess its feasibility and automatically generate test data that causes its execution. For each path the system extracts the path predicates and the assignment statements. The statements on the path are then symbolically executed to produce a path condition and expressions for each of the output variables. The path condition is NOT checked for feasibility as each predicate is added but is invoked only once when the PC is complete resulting in faster feasibility checking. When a solution is found the path is feasible and the solution may be used as a test case. The solution is placed as records in the appropriate files making the test execution of the program a simple task. The initial path extraction mechanism allowing input of source program line numbers is not yet implemented and must therefore be undertaken using the 'user path selection' mechanism of the system.

7.4.3 Debugging tool

During testing the user often constructs functional test cases from the knowledge of what the software is expected to do. The program is then executed with the test cases using dynamic analysis monitoring facilities which provide an execution history of the test cases allowing various coverage metrics to be assessed. Additional test cases may then be devised to increase

the level of coverage. The program is executed with the new test cases. This cycle is repeated until the coverage cannot be increased.

When the cause of an error case is easily identified, corrections are made and testing repeated. However, when the cause is not easily pinpointed the SYM-BOL system may be used. The path exercised by the test case which revealed the error is symbolically executed. After the execution of each statement the user is provided with diagnostic output showing the symbolic value of each variable. This information is much more useful than the trace table provided in most COBOL compilation systems because it provides visibility of the expression that yields the erroneous result.

7.4.4 Branch-path location

During the testing process there may be some difficulty in manually deriving test data to execute some branches. Branches which are not covered by previous tests need to be located on a feasible path if they are to be executed. Finding such a feasible path may not always be straightforward primarily because of the care that needs to be taken in undertaking the search by hand. The system can be used to help in this task by allowing commencement from an existing feasible path.

The strategy is to start with the feasible path closest to the branch that is to be executed and to divert this path to the branch to be included. Automating a repeated search based on existing feasible paths is a relatively difficult process because the number of paths that could

be searched is large and there are no obvious search reduction strategies. The first version of the system does not include this feature but a simple mechanism which allows a feasible path to be used as a starting point but does not include a means of repeatedly searching until no more feasible paths are available is to be included in a second version of the system.

7.4.5 SYM-BOL, 'ideal' and EFFIGY compared

EFFIGY was identified in chapter 5 as a suitable benchmark for assessing future symbolic execution testing systems. The EFFIGY system provides the following features:

* input of source code containing assertions;
* interactive selection of paths;
* output of (a) symbolic values,
  (b) truth of assertions,
  (c) statement on path feasibility.

SYM-BOL provides all of these features. Additionally, SYM-BOL processes programs written in COBOL, a language more widely used than PL/1, the language processed by EFFIGY. The SYM-BOL system caters for most COBOL features with the exception of indexed files and the string handling verbs. This is in contrast with EFFIGY which processes a comparatively small subset of PL/1. Appendix B contains full lists of allowed and barred features. A method of processing the string handling verbs has been devised but as yet this has not been implemented. This is described in Chapter 12.

Tables 7.1a&b, 7.2a&b and 7.3a&b show an extended version of tables 5.1, 5.2 and 5.3. The new tables include the SYM-BOL system, the seven systems that satisfy the minimum criteria and the other systems that claim some use of symbolic execution. The systems have been classified in terms of how well they meet the criteria established for the ideal system. This is highly subjective requiring the assessment of the significance of each of the criteria as well as of how well each system meets these criteria.

Two significant factors not listed in the tables are the contribution to research and the practicality of the tool. For example, EFFIGY has the edge over the FORTRAN testbed in terms of contribution to research. EFFIGY includes all the features expected of a symbolic execution testing system, with the exception of test case generation, whereas the FORTRAN testbed does not cater for assertions nor does it incorporate a path selection strategy or provide output of variable expressions and the path condition at each branch point. On the other hand, EFFIGY is not such a generally useful tool as it can be applied only to a limited language subset, whereas the FORTRAN testbed can process the full language specification and it is a commercial product demonstrating the practicality of the tool and its techniques.

SYM-BOL includes all of the features included in EFFIGY. It analyses a language and a class of programs previously ignored by symbolic execution research. However, it is a prototype that is far from robust and in need of much further work before it could become a commercial product. Attempting to rank these three systems is thus difficult. Tables 7.1a&b, 7.2a&b and 7.3a&b classify all the systems into ranked bands. The systems in each band are deemed equally meritorious even though they may show quite different characteristics.

System	Author	Date	Language built in	Language analyzed	Input	Output
Ideal system			Widely available	Many widely available languages	Source program, assertions	Path, PC, Point becomes infeas Output variable expressions, Truth of assertions, Coverage Test data, Active/idle domains
LEVEL ONE						
EFFIGY	King	1975	not stated	Simple PL/1	Source program, assertions	Symbolic vals, Truth of assertions
IPS	Asirelli	1979	PL/1	Subset PL/1	Source program, assertions,	Symbolic expressions, Path condition, Truth of assertions
FORTRAN Test-bed	Hedley	1981	ALGOL 68 FORTRAN	FORTRAN	Source program, paths	Test cases, Infeasible paths
SYMBAD	Coen-Parisi	1990	C and LISP	Ada sequential units only	Source program, assertions	Symbolic expressions, Path condition, State of assertions
SYM-BOL	Coward	1991	COBOL	COBOL excluding: indexed files, string verbs reference modification	Source program, assertions	Path, PC, point becomes infeas Variable expressions, Truth of Coverage Test cases
LEVEL TWO						
SELECT	Boyer	1975	LISP	subset LISP	Source program, Assertions	Not stated
CASEGEN	Ramamoorthy	1976	FORTRAN	FORTRAN	Source program	Paths, Test cases
ATTEST	Clarke	1976	FORTRAN	FORTRAN	Paths	Symbolic expressions, Test cases

Table 7.1a Symbolic Execution Systems Compared

System	Author	Date	Language built in	Language analyzed	Input	Output
LEVEL THREE						
SMOTL	Bicevskis	1979	Unknown	SMOD	Source program	Minimal set of test cases giving branch coverage
LEVEL FOUR						
DISSECT	Howden	1977	LISP	FORTRAN	Source program, DISSECT commands	Path in straight line form, Path condition, Var expressions, Anomalies
SADAT	Voges	1980	Pl/1	FORTRAN	Source program SADAT commands	Path conditions for set of paths to cover all branches, some may be infeasible
IVTS	Taylor	1983	Pascal	HAL/S	Source program	Anomalies, Path conditions
UNISEX	Kemmerer	1985	YACC, LISP	Pascal excludes: files, pointers, boolean operators	Source program containing assertions	Reformatted source, PC, Truth of assertions
LEVEL FIVE						
EL1	Cheatham	1979	unknown	EL1 subset	Source program	Intermediate representation

Table 7.1b Symbolic Execution Systems Compared

System	Automatic/ Interactive	Path selection	Path feasibility	Assertion testing
Ideal system	Automatic and interactive	Uses symbolic expressions to aid in selection	Linear and non-linear	Yes
LEVEL ONE				
EFFIGY	Interactive	User selected	Every branch, Theorem prover	Yes
IPS	Interactive	User control	Every branch, Theorem prover	Yes
FORTRAN test-bed	Automatic	Not stated	Every branch, Algebraic linear and random	No
SYMBAD	Interactive	User selected	Every branch, theorem prover, linear	Yes
SYM-BOL	Automatic and interactive	User or automatic, Uses symbolic expressions to aid in selection	Every branch, Algebraic linear, detects non-linear	Yes
LEVEL TWO				
SELECT	Automatic	All paths	Every branch Algebraic linear and non-linear	Yes
CASEGEN	Automatic	Minimal set cover all branches	Every branch Algebraic linear non-linear and random	No
ATTEST	Automatic and interactive	None	Every branch Algebraic only linear	No

Table 7.2a Symbolic Execution Systems Compared

System	Automatic/ Interactive	Path selection	Path feasibility	Assertion testing
LEVEL THREE				
SMOTL	Automatic	Connection of sub-paths, Reduction strategy	Assessed from min & max values of variables	No
LEVEL FOUR				
DISSECT	Batch	User selects	User by hand	No
SADAT	Batch	User selects	User by hand	No
IVTS	Interactive	User selects	User by hand	Yes
UNISEX	Automatic and interactive	True branch first or user selects	User by hand	Yes
LEVEL FIVE				
EL1	Automatic	Unknown	Not undertaken	No

Table 7.2b Symbolic Execution Systems Compared

System	Call Handling	Loop Handling	Array Handling	File Handling	String Handling
Ideal system	User choice between macro-expan and I/O boundary	0,1,2 iterations plus minimum to give branch coverage	Maintain symbolic expressions with no path explosion	Maintain symbolic expressions by file, Output files of test cases and expected results	Maintain symbolic expressions for each charc, Assess feas string constants
LEVEL ONE					
EFFIGY	I/O boundary	User selects	1-dim	No	No
IPS	Not stated	User selects	Not known	No	No
FORTRAN test-bed	Macro-expan, functions as I/O boundary	User specifies number of iterations	Maintains symbolic expressions	No	No
SYMBAD	Not stated	User selects	Yes	No	No
SYM-BOL	Macro-expan and I/O boundary	User selects or minimum number for coverage	Uses actual value	Maintain symbolic expressions by file, Output files of test cases + expected results	Assesses feasibility for path conditions containing string constants
LEVEL TWO					
SELECT	Macro-expan	User specify max no of iterations	Yes	No	No
CASEGEN	Not stated	Fixed number	Yes	No	No
ATTEST	Macro-expan	Fixed max no of iterations	only constant indexes	No	No

Table 7.3a Symbolic Execution Systems Compared

System	Call Handling	Loop Handling	Array Handling	File Handling	String Handling
LEVEL THREE					
SMOTL	Unknown	minimum iterations	Not a problem no symbolic expressions	Yes	No
LEVEL FOUR					
DISSECT	Not stated	User selects	Virtual paths, or maintain ambiguity, or actual values	No	No
SADAT	Not stated	Not stated	Not stated	Not stated	No
IVTS	Not stated	Not stated	Not stated	Not stated	No
UNISEX	I/O boundary user	0,1 and selects	Virtual paths	Excluded	No
LEVEL FIVE					
EL1	I/O boundary	Solves some recurrence relations	Unknown	No	No

Table 7.3b Symbolic Execution Systems Compared

When SYM-BOL is compared with the seven systems that satisfy the three minimum criteria for a symbolic execution system it stands up well. The top two bands in the tables contain the seven systems that satisfy the three minimum criteria for a symbolic execution system together with SYM-BOL. SYM-BOL is placed in the top band along with EFFIGY, the FORTRAN testbed, IPS and SYMBAD, ahead of SELECT, CASEGEN and ATTEST which are placed in the second band. When SYM-BOL is compared with all thirteen of the systems claiming to undertake symbolic execution it appears favourably and is a significant contribution to research in the field. The following brief discussion compares SYM-BOL against only the seven systems that satisfy the minimum criteria and ignores the systems that fail to meet the minimum criteria except where they contain a useful feature not included in the seven systems.

Path selection in SYM-BOL can be carried out either by user selection or automatically. This is in contrast to ATTEST and the FORTRAN testbed which report no mechanism for path selection other than the input of paths. The remaining systems provide either automatic or user path selection but not both.

When the user is selecting paths SYM-BOL aids the selection by displaying the current expressions for each variable in the branch predicates. This helps the user avoid the selection of obviously infeasible paths. Also displayed is a branch coverage measure to help the user select branches not covered by a path. This is not provided by the other systems which allow user selection of paths. IPS and EFFIGY do however allow the display of variable expressions

at each node.

When the SYM-BOL system is used to automatically generate paths the branch coverage measures and variable expressions are used in a bid to select feasible paths containing previously uncovered branches. In addition, when faced with otherwise equal choices between branches the path selection chooses the branch that gives the greatest domain coverage. At present this domain maximizing selection is rather crude but it is planned to be developed in a later version. No other symbolic execution testing system incorporates such a strategy. The basis of the approach is that used in SMOTL to determine path feasibility. Minimum and maximum values are maintained for each variable. The branch that provides the greatest distance between the minimum and maximum values is the one selected when in all other respects the alternative branches are equally appropriate.

A powerful feature of SYM-BOL's path selection facilities is the inter-relationship between the symbolic execution, which creates variable expressions, and the selection of branches for a path. The benefit from this is a reduction in the selection of infeasible paths. Only EFFIGY and the IPS provide variable expressions to help the user in making the selection. ATTEST and the FORTRAN testbed have no path selection strategy specified paths being an input to the system. SELECT and CASEGEN aim for branch coverage whilst path selection in EFFIGY, IPS and SYMBAD is solely user driven.

First indications in using SYM-BOL are that the most useful mode is the user selection as this

overcomes the branch selection problem of what might be termed 'busy-bottlenecks'. Here, a small number of branches must be executed many times to allow all the branches beyond these points to be executed at least once. Busy-bottlenecks are easily identified and handled by the user but no method is incorporated within the automatic mode of SYM-BOL, nor any other automatic system, to deal with this problematic, yet common, situation.

When compared to the ideal system described in Chapter 5, SYM-BOL fails to meet the ideal criteria because it can process only COBOL programs and there are some restrictions within the COBOL language. It deals with ambiguous array references by substituting an actual value in place of the input variable index. This is far from satisfactory but is a practical way of overcoming a difficulty which is caused by the need for a run-time value during a static analysis. SYM-BOL does, however, meet many of the ideal system's requirements. It allows both automatic and interactive use, the source program may contain assertions which can be verified and all of the required outputs with the exception of an idle/active domain report are provided. SYM-BOL also uses the products of symbolic execution to aid path selection in choosing the next branch. This is preferable to making a random branch choice and then testing for feasibility repeatedly until a feasible branch is selected.

INTRODUCTION TO CHAPTERS EIGHT TO ELEVEN

The next four chapters describe aspects of the functioning of the SYM-BOL system. Where a useful feature has been designed but not implemented this is stated in the text.

Chapter eight describes how the system processes assertions placed in the source program. The assertions need not be deleted from the final program. If so desired they may be transparent from the compiler. Alternatively, they can be included in the final executable code.

Chapter nine describes a series of translations which are made to the COBOL source program. Several stages of translation into simpler standard forms are employed. The final version of the 'source' program is then translated into the intermediate form.

Chapter ten explains the approach to path selection and symbolic execution. Early systems undertook path selection and symbolic execution in isolation. By synchronizing the two processes the results of symbolic execution are used to help make sensible path selection choices. SYM-BOL uses this co-ordinated approach to path selection in both automatic and user path selection modes.

Chapter eleven describes why the NAG library linear optimizer E04MBF is suitable for path feasibility checking and how it is used. Two problems, concerning alphanumeric literals (string constants) and implied constraints in record structures, are outlined together with the means of overcoming them.

CHAPTER EIGHT ASSERTIONS

Assertions are used in the axiomatic approach to program proving. This approach requires that the program validator is able to make statements about the values of variables at various points in the software. Such a statement about what a variable should contain is an assertion.

8.1 Introduction

To prove that the program statements between two points are correct it must be shown that given that the initial assertions are true then the statements up to the end point cause the final assertions to be true. Such a demonstration will have proved partial correctness. Complete correctness requires that it can be shown that the program halts and hence there is no endless loop. To prove a program partially correct requires at least two sets of assertions, one at the beginning and one at the end of the program.

Symbolic execution can contribute towards this form of program proving [Hant76]. Assertions at any point in a program are conjoined onto the path condition. If the path condition containing the assertions is feasible then the assertions are upheld. An alternative approach is to negate the assertions before conjoining to the path condition. If the path condition is feasible then the assertions are not upheld. The SYM-BOL system adopts the former approach.

The issue of complete correctness versus partial correctness is not such a major one for symbolic execution because the assessment of correctness is applied to a path. Each path has a terminating point. When an infinite loop exists in the program then the only paths that pass

through the loop will be infeasible. Inability to find a feasible path through a loop would suggest that the program is not completely correct.

This chapter describes the assertion checking facilities available in SYM-BOL.

8.2 Specifying assertions in SYM-BOL

Assertions may be included in the source program at any point. The format adopted for their specification is as follows:

```
*assert
*    condition-1
*    condition-n
*end-assert
```

For example in a program processing salaries the initial and final assertions might be as follows:

```
*initial assertions
*assert
*    (gross    > -1) and
*    (gross    < 10000) and
*    (tax-code > -1) and
*    (tax-code < 10000)
*end-assert

*final assertions
*assert
*    ((state = "e") or (state = "n")) and
*    ((tax < gross) or (tax = gross)) and
*    ((ni < gross)  or (ni = gross))  and
*    ((tax > 0)     or (tax = 0))     and
*    ((ni > 0)      or (ni = 0))
*end-assert
```

Note that the assertions are written as comment statements so that normal compilation, linking and execution, are unaffected by their presence. By utilizing the assertion processing option in the SYM-BOL system assertions are transformed into equivalent 'evaluate' statements as follows:

```
*initial assertions
*assert
     evaluate true
       when gross > -1          continue
     end-evaluate
     evaluate true
       when gross < 10000       continue
     end-evaluate
     evaluate true
         when tax-code > -1     continue
     end-evaluate
     evaluate true
         when tax-code > 1000   continue
     end-evaluate
*end-assert

*final assertions
*assert
     evaluate true
       when state = "e"         continue
       when state = "n"         continue
     end-evaluate
     evaluate true
         when tax < gross       continue
         when tax = gross       continue
     end-evaluate
     evaluate true
         when ni < gross        continue
         when ni = gross        continue
     end-evaluate
     evaluate true
         when tax > 0           continue
         when tax = 0           continue
     end-evaluate
     evaluate true
         when ni > 0            continue
         when ni = 0            continue
     end-evaluate
 *end-assert
```

The action of compilation now checks the syntax of both the assertions and the rest of the program. Note that the imperative statement in this form of the assertions is 'continue', no action is taken. When the program contains this version of the assertions the program still executes in the usual way and provides the same results.

Assertions can be used to provide a 'dynamic semantic monitor' by negating the assertions and changing the resulting 'continue' statements into error messages which report when an assertion violation has occurred.

```
Identification Division.
Program-Id. P9a.
Environment Division.
Data Division.
Working-Storage Section.
01  ni-details.
    02   ni-rate          pic  99v99.
01  tax-details.
    02   tax-free         pic  9(4)v99.
    02   tax-rate         pic  99v9.
    02   taxable          pic  9(5)v99.
Linkage section.
01  input-parameters.
    02   gross            pic  9(5)v99.
    02   tax-code         pic  9(4).
    02   ni-class         pic  x.
    02   frequency        pic  x.
         88   weekly      value "w".
         88   monthly     value "m".
01  output-parameters.
    02   state            pic  x.
         88   err         value "e".
         88   no-error    value "n".
    02   tax              pic  9(4)v99.
    02   ni               pic  9(3)v99.
Procedure division using input-parameters output-parameters.
the-program.

*initial assertions  Ai
*assert
*    (gross    > -1) and
*    (gross    < 10000) and
*    (tax-code > -1) and
*    (tax-code < 10000) and
*    ((ni-class = "a") or (ni-class = "b") or (ni-class = "c")) and
*    ((frequency = "w") or (frequency = "m"))
*end-assert

*initialize
     move "n" to state.

*set the ni rate
     evaluate true
       when ni-class = "a"     move  0.05  to ni-rate
       when ni-class = "b"     move  0.10  to ni-rate
       when ni-class = "c"     move  0.15  to ni-rate
       when  other             move  0.0   to ni-rate
                               move "e" to state
                               display "ni class error"
     end-evaluate

*A1
*assert
*    ((ni-rate > 0) or (ni-rate = 0)) and
*    (ni-rate < 0.2)
*end-assert
```

Figure 8.1 Program containing assertions

```
*set tax free
     if state not = "e"
     then
       if monthly
       then
         compute tax-free = tax-code * 10 / 12 end-compute
       end-if
       if weekly
       then
         compute   tax-free = tax-code * 10 / 52 end-compute
       end-if
       if not monthly and not weekly
       then
         move 0 to tax-free
         move "e" to state
         display "error in pay period"
       end-if
     end-if
*A2
*assert
*    ((tax-free = 0) or (tax-free > 0)) and
*    (tax-free < taxcode)
*end-assert
*set tax rate
     if state not = "e"
     then
       compute  taxable = gross - tax-free  end-compute
       if taxable < 10000
       then
         move 0.3 to tax-rate
       else
         if taxable < 20000
         then
           move 0.4 to tax-rate
         else
           move 0.5 to tax-rate
         end-if
       end-if
*A3
*assert
*    ((taxable < gross) or (taxable = gross)) and
*    (tax-rate > 0) and
*    (tax-rate < 1)
*end-assert
* calculate deductions
       compute ni   = ni-rate * gross  end-compute
       compute tax  = tax-rate * taxable  end-compute
     end-if
*final assertions  Af
*assert
*    ((state = "e") or (state = "n")) and
*    ((tax < gross) or (tax = gross)) and
*    ((ni < gross)  or (ni = gross))  and
*    ((tax > 0)     or (tax = 0))     and
*    ((ni > 0)      or (ni = 0))
*end-assert
     exit program.
 end program P9a.
```

Figure 8.1 Program containing assertions (continued)

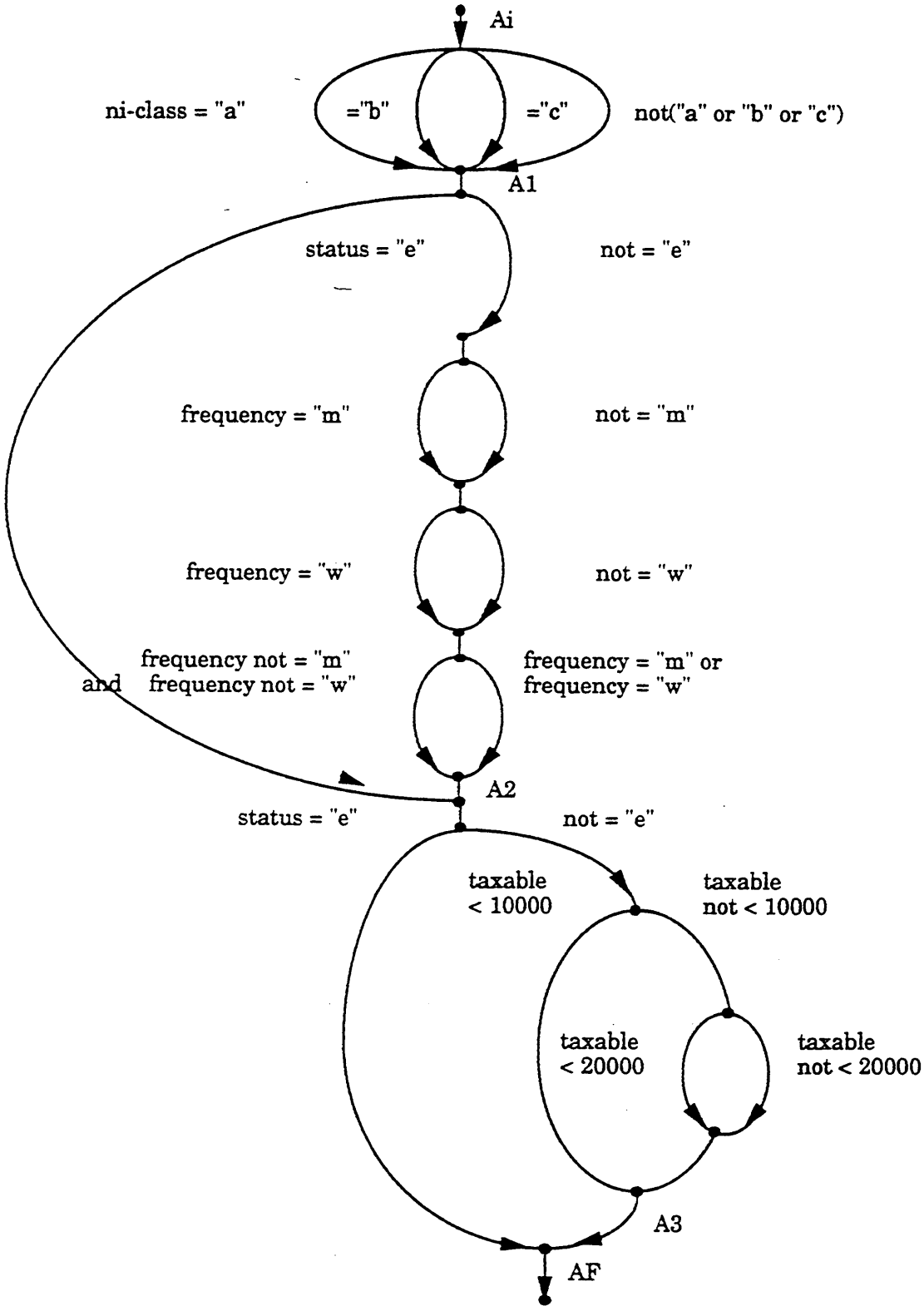

Figure 8.2 Flowgraph of program containing assertions

8.3 Assertions and general proofs

When a path containing assertions is found to be feasible the assertions are deemed proven. Unfortunately, the proof is not general because it applies only to the path examined. The same assertions may occur on other paths yet may not be upheld. To obtain a general proof requires the symbolic execution of all paths between the two points. If these points are the start and end of the program this is usually impossible due to the existence of a large, and often infinite, number of paths. This is the perennial path-based testing weakness. To minimize this effect the number of points at which assertions are placed in a program should be increased. The closer the two points between which the proof is attempted the fewer the number of partial-paths that exist between them.

A proof is attempted for all partial-paths between every consecutive pair of sets of assertions. With the exception of the initial and final assertions each set of assertions is used as both initial and final assertions in assessing feasibility of partial-paths.

Consider the program in figure 8.1. Notice that it has three sets of intermediate assertions as well as initial and final assertions. Figure 8.2 shows a flowgraph of the program in figure 8.1. There are 144 paths through this program. If only initial and final assertions were included all 144 paths would need to be symbolically executed for a proof to have been achieved for the program. The inclusion of intermediate assertions reduces this number significantly as follows:

Assertion Pairs	Number of Partial-paths
Ai - A1	4
A1 - A2	9
A2 - A3	3
A2 - Af	1
A3 - Af	1
	=
	18

All intermediate assertions are used as both initial and final assertions thus ensuring connection of all paths from the beginning to the end of the program. All assertion to assertion partial-paths must be symbolically executed and the final assertions verified correct. Consider the assertion points A1 and A2. There are nine partial-paths between these assertion point pairs. All nine partial-paths are symbolically executed and deemed correct. A similar verification is undertaken for assertion point pairs A2-A3, A2-Af and Ai-A1. Because the assertions at A1 are used as final assertions in verifying Ai-A1 we can be certain that the program is correct at point A1. Assertions at A1 are also used as initial assertions in verifying A1-A2. This pairing is deemed correct given that A1 is correct. We already know that A1 is correct from the previous verification stage. This process is repeated for all adjacent assertion point pairs. The result is that all pairs of consecutive assertions are verified and thus, given that the initial assertions are true, then the program will always satisfy the final assertions.

Looping introduces the problem of termination. Assertions are placed immediately before and immediately after the loop. The difficulty is that there is often a large, even infinite, number of possible iterations. A common approach is to consider a maximum of two iterations together with just one iteration and in the case of a pre-condition loop a zero iteration. On each iteration all possible partial-paths through the loop are verified. For the case containing

two iterations this would appear to require the consideration of the square of the number of partial paths through the loop. This can be avoided by placing assertions at the beginning of the loop yet inside the loop and at the end inside the loop. Each iteration of the loop now has a set of initial and final assertions. By making these assertions identical the verification of the final assertions establishes the truth of the initial assertions for the next iteration. In principle two iterations are sufficient to verify the loop.

The key to this approach is that all modified variables on a partial-path must be the subject of strict assertions. An omission will invalidate the verification; so too will weak assertions that are easily satisfied. Provided there are sufficient strict, well placed, sets of assertions in a program symbolic execution is a powerful verification tool.

The SYM-BOL system allows the inclusion of assertions at any point in a program. By selecting the assertion option they are transformed into equivalent COBOL statements. As path selection proceeds assertions are conjoined to the path condition and at feasibility checking are verified. When feasibility checking is requested at every branch it is known immediately which predicate, whether program code or assertion, caused infeasibility. At present the system does not cater for assertion-point to assertion-point verification.

8.4 General assertions

The assertions discussed so far are of a simple type such as:

```
a > b
```

It is often possible and desirable to express far more wide-ranging assertions than ones stating a relationship between two simple variables. For example the assertion:

```
for all i where i in 1 to max : a[i] > 0
```

stipulates that all elements of the array 'a' contain values greater than zero. This can be expressed as a series of simple assertions of the form:

```
a[1] > 0
a[2] > 0
etc.
```

It is clearly impractical to expect users of SYM-BOL to write such assertions in such a long specific form so a translation from general to specific forms is required as a function in the system.

A similar strategy is to be adopted for assertions about files. For example:

```
for all i where i in 1 to max - 1 : file.key[i] < file.key[i + 1]
```

asserts that the file is in ascending order on the field 'key'.

Programs containing assertions about many large arrays and files will be faced with a large number of simple assertions resulting in optimization problems with a large number of variables. In comparison with complex optimization problems these are quite straightforward but the concern for a symbolic executor is the response time for feasibility checking. This is

not likely to be a problem but further work is required to establish the bounds for acceptable run-time of the system. The current version of SYM-BOL does not cater for general assertions.

8.5 Summary

Three of the seven existing systems, ATTEST, the FORTRAN testbed and CASEGEN, do not cater for the verification of assertions. Four of the systems assess path feasibility using optimization techniques the other three use theorem provers. Only SELECT uses optimization techniques for feasibility checking and verifies assertions.

Creation of the SYM-BOL prototype increases to two the number of systems which use optimization techniques to assess path feasibility and provide assertion verification, though this is not a particularly significant combination. Assertions are encoded as comments within the source program. This allows normal compilation and execution to be unaffected by their presence, yet they are a permanent feature of the program documentation. A simple transformation of the assertions into equivalent COBOL statements suitable for inclusion on the path condition, allows a simple means of verifying the assertions. Only MALPAS and SYMBAD maintain assertions as permanent features of the source program by specifying assertions as comments.

CHAPTER NINE TRANSFORMING A SOURCE PROGRAM INTO INTERMEDIATE FORM

Symbolic execution is usually achieved by translating the source program into an intermediate representation which is designed to be more appropriate than the source program for extracting and symbolically executing a path.

The creation of an intermediate form can be achieved in two ways. First, it can be created by direct translation from the source program. This requires a complex translator. Second, the source program can undergo several steps of translation into simpler or standard forms in the source language. The final version of the 'source' program is then translated into the intermediate form by a comparatively simple translator. The SYM-BOL system uses the second of these two approaches.

This chapter describes a series of translations which are made to the COBOL source program. These translations can be grouped into: alphanumeric literals; condition names; assignment constructs; and branching constructs. A discussion of alphanumeric literals is postponed until chapter 11 where it appears as part of the discussion on formulating path conditions as LP problems.

9.1 Condition names

A condition name is a high-level means of expressing a conditional expression. Consider the following variable declarations:

```
01  birthday              pic x.
    88 birthday-past      value "p".
    88 birthday-today     value "t".
    88 birthday-later     value "l".
```

A conditional statement using a condition name would be of the form:

```
if birthday-today
then...
```

This is equivalent to:

```
if birthday = "t"
then...
```

Similarly,

```
set birthday-today to true
```

is equivalent to:

```
move "t" to birthday.
```

To enable conditional statements containing condition names to be treated in the same way as other conditions the simplest approach is to substitute the condition name with the full lower-level condition which it represents and to transform 'set' statements into equivalent 'move' statements.

9.2 Assignment constructs

These can be categorized into: string processing; simple assignments; arithmetic constructs; input statements; and module invocation.

9.2.1 String processing

Simple string manipulation, such as assigning a string to a variable, is carried out using the 'move' construct. COBOL provides the verbs: 'inspect'; 'string'; and 'unstring' together with reference modification to provide more sophisticated facilities for string manipulation. These more sophisticated facilities are not currently handled by the SYM-BOL system. However, the system does include the use of strings in conditional expressions which are handled in a

novel way described later in chapter 11. The system also includes the use of simple string assignments using the 'move' verb. The problems presented by the more sophisticated string manipulation and the means of catering for these in symbolic execution are discussed in chapter 12.

9.2.2 Simple assignments

The COBOL verbs: 'move'; and 'set' are the facilities provided in COBOL for assigning the contents of one variable to another. 'Move' is adopted as the standard form. The 'set' construct is used for the same purposes as 'move' except that it can be used to act only on index variables and level-88 condition names. Where it is used on index variables the declaration will need to be modified as well as the set statement if it is to be transformed into a move statement. For example consider the following:

```
01  table.
    03 row occurs 100 indexed row-index.
       05 field1  pic x.
       05 field2  pic x.

     set row-index to 1
```

To transform the set statement into a move statement also requires the creation of row-index as an independent data item. The resulting transformation is as follows.

```
01  table.
    03 row occurs 100.
       05 field1  pic x.
       05 field2  pic x.
01  row-index     pic 999.

     move 1 to row-index
```

9.2.3 Arithmetic constructs

The COBOL verbs: 'add'; 'subtract'; 'multiply'; 'divide'; 'compute'; 'set'; and arguably "perform varying' are provided in COBOL for evaluating arithmetic expressions and assigning the result to a variable. All of these can be expressed using 'compute'. For example consider:

1.
```
add a to b
```
is transformed to:
```
compute a = a + b
```

2.
```
perform varying a from 1 by 1 until a = 10
  continue
end-perform
```
is transformed to:
```
move 1 to a
perform until a = 10
  continue
  compute a = a + 1
end-perform
```

9.2.4 Input statements

'Accept'; and 'read' are the facilities provided in COBOL for input of data into variables. 'Accept' is used to set input variables interactively and 'read' is used to extract data from files. No standard form is adopted for this group. During symbolic execution both 'read' and 'accept' cause the assignment of a new symbolic value.

9.2.5 Module invocation

The COBOL verb: 'call'; is the means of invoking a sub-program. The verb 'sort' could also be included in this class. Although it is a COBOL provided facility 'sort' behaves in much the same way as 'call'. Invoking 'sort' requires the passing of a file and the return of a new file.

When treating a module call as an output-input statement the returned results are given new symbolic values. 'Call' can be treated as an output-input statement or as a 'perform' causing symbolic execution to continue into the called routine. 'Sort' is always treated as an output-input statement.

9.3 Branching constructs

The transformations made to branching constructs are necessary to facilitate the creation of a path condition and its subsequent feasibility checking. Path conditions are made up of a set of conjoined predicates taken from the conditions at each branch point on a path. When the condition is a simple two-way selection either the predicate expressed in the condition (the true branch) or the negation of the predicate expressed in the condition (the false branch) is conjoined to the path condition. The predicates that make up the path condition are used to form a set of constraints in a linear programming (LP) problem which is submitted to an optimizer.

9.3.1 Simple conditions

It is a simple matter to formulate simple equalities into a constraint appropriate for submission to an optimizer. However, it is a little more difficult to formulate the negated condition into suitable constraints.

Consider the condition: if a = b

The predicate for the true branch is: a = b.

The predicate for the false branch is: a <> b.

The predicate for the true branch can be passed to an optimizer but the predicate for the false branch cannot. The only way of passing a negated condition to the optimizer is to express it as a compound condition: a < b or a > b. However, if both of these constraints are passed to the optimizer then the problem will not be solvable due to a contradiction in the constraints. Each of the predicate components are implicitly separate branches.

One way of overcoming this difficulty is to transform all apparently two-way selections into three-way selections, each branch representing one of the three possible predicates that can be passed as a constraint to the optimizer. For example:

```
if a = b                                    evaluate true
then                                            when a < b  s2
   s1                                           when a = b  s1
else          can be expressed as:              when a > b  s2
   s2                                       end-evaluate
end-if
```

Each of the three constraints a < b, a = b, a > b, can be successfully passed to an optimizer as a constraint on a path condition.

9.3.2 Compound conditions

Where the predicates contain boolean operators forming compound conditions a similar problem arises to the one facing the false branch of a simple condition. There is no difficulty when a branch predicate contains only the boolean operator AND. Here, the component

conditions of the compound condition are treated as individual predicates and constitute two or more constraints. However, the negation of this compound condition cannot be passed as a constraint. Consider the conditional statement:

```
if a = b and c = d
then
   s1
else
   s2
end-if
```

The true branch predicate:

```
a = b and c = d
```

can be conjoined to the path condition and passed to the optimizer as two constraints:

```
1  a = b,
2  c = d.
```

The false branch predicate:

```
not (a = b and c = d)
```

which can be expressed as:

```
a <> b or c <> d
```

cannot be passed as a constraint. One approach is to decompose the compound condition to produce nested if statements which in turn can be transformed into evaluate statements containing appropriate constraints. For example:

```
if a = b                                  evaluate true
then                                        when  a = b
   if c = d                                   evaluate true
   then                                         when c > d   s2
      s1      can be expressed as               when c < d   s2
   else                                         when c = d   s1
      s2                                      end-evaluate
   end-if                                   when a > b       s2
else                                        when a < b       s2
   s2                                     end-evaluate
end-if
```

An alternative approach is to use a single evaluate statement which is, essentially, a form of truth table:

```
evaluate true also true
  when   a<b  also c<d  s2
  when   a=b  also c<d  s2
  when   a>b  also c<d  s2
  when   a<b  also c=d  s2
  when   a=b  also c=d  s1
  when   a>b  also c=d  s2
  when   a<b  also c>d  s2
  when   a=b  also c>d  s2
  when   a>b  also c>d  s2
end-evaluate
```

Each of the WHEN clauses forms part of a path condition which can be passed to an optimizer as a system of constraints. There is no need for further negation of conditions to take place as all possible conditions are explicitly specified in the newly constructed evaluate statement(s).

An alternative to transformation into nested simple conditions and truth tables requires the maintenance of several path conditions for each path. The precise number of path conditions varies with the number of OR operators. At the point on the path where a compound condition containing OR is encountered the maintenance of two or more path conditions is commenced. The first PC contains the simple condition before the OR operator, the second PC contains the simple condition after the OR operator. This approach has the benefit that it maintains the structure of the program that was created by the programmer. Having multiple PCs for each path increases the number of PCs that are to be checked for feasibility. This is no worse than creating additional paths by expanding the compound conditions. However, there is an overhead in applying this technique. The symbolic execution component of the testing system must maintain the connections between the multiple PCs and their paths. Whereas the use of compound condition expansion needs no such mechanism and a dynamic analysis coverage monitor can be used to assess the impact of the testing.

The SYM-BOL system is designed to use compound condition expansion to deal with conditions containing the OR logical operator.

9.3.3 WHEN OTHER exit from multi-branch conditionals

Consider the following program fragment:

```
evaluate true
  when   x = a   s1
  when   x = b   s2
  when   other   s3
end-evaluate
```

The predicates for the first two cases are straightforward: x = a, x = b. The third case is more complex being: x <> a and x <> b. This compound condition is transformed in a manner similar to that described above for compound conditions. The following shows an appropriate transformation:

```
evaluate true
  when x = a          s1
  when x = b          s2
  when x < a
    evaluate true
      when x < b      s3
      when x > b      s3
    end-evaluate
  when x > a
    evaluate true
      when x < b      s3
      when x > b      s3
    end-evaluate
end-evaluate
```

When the conditions in the multi-branch construct contain constants the transformation can be to a simpler form. Consider the following program fragment:

```
evaluate true
  when   x = "a"   s1
  when   x = "c"   s2
  when   other     s3
end-evaluate
```

The following shows its simpler transformation:

```
evaluate true
  when   x = "a"                s1
  when   x = "c"                s2
  when   x < "a"                s3
  when   x > "a" and x < "c"    s3
  when   x > "c"                s3
end-evaluate
```

This transformation requires all missing domains to be included in the list of alternatives. This is easily achieved by arranging the constants in sort sequence allowing easy specification of the domain constraints. It may be that some of the derived domains are empty. Consider the following declaration:

```
01  x  pic x.
```

Now consider the domain defined as:

```
when x > "a" and x < "b"
```

This is empty and can be deleted.

9.3.4 Iterations

Iterations of the form:

```
perform test before until a = b
  s1
end-perform
```

give rise to conditions which provide constraints for inclusion on the PC. This may be achieved by transforming into the following form.

```
 1   iter1.
 2
 3       evaluate true
 4           when a<b    continue
 5           when a=b    go end-iter1
 6           when a>b    continue
 7       end-evaluate
 8       s1
 9       go iter1.
10   end-iter1.
```

Should the loop be of the form:

```
perform test after until a = b,
```

the statements at line 8 are moved to line 2. Where the condition is a compound condition the transformation is of the same form but in this case has two evaluate statements. For example:

```
    perform x test after
      until a = b or c = d
    s1.
x.
    s2
    s3.
```

is transformed into:

```
iter1.
    perform x
    evaluate true
      when a = b   go end-iter1
      when a < b   continue
      when a > b   continue
    end-evaluate

    evaluate true
      when c = d   go end-iter1
      when c < d   continue
      when c > d   continue
    end-evaluate
    go iter1.
end-iter1.
    s1.
x.
    s2
    s3.
```

9.3.5 Compute - on size error - not on size error

Many statements contain branches which can be converted to the evaluate standard form. For example, in the following program fragment, by using the picture declaration of variable 'a' to provide the maximum value allowed for 'a' the 'size error' clauses can be replaced.

```
a pic 99.
b pic 99.
c pic 99.
compute a = b + c
  size error      s1
  not size error  s2
end-compute
```

```
a pic 99.
b pic 99.
c pic 99.
compute a = b + c
evaluate true
  when b + c < 99    s2
  when b + c = 99    s2
  when b + c > 99    s1
end-evaluate
```

9.3.6 Read - at end - not at end

Input and output statements also contain branching. Consider the following program fragment:

```
data division.
file section.
fd  fa-in record varying depending wa-length.
01  fa-rec          pic x(50).
working-storage section.
01  wa-length       pic 9(5) comp.
01  wa-in           pic x(50).
procedure division.
theprogram.
    read fa-in into wa-in
        end       s1
        not end   s2
    end-read
    s3
```

To transform this into the evaluate standard form requires the use of declarative sections to remove the 'end' and 'not end' clauses. Statements in the declarative sections are not conjoined onto the path condition. The transformed program fragment is as follows:

```
data division.
file section.
fd  fa-in record varying depending wa-length.
01  fa-rec          pic x(50).
working-storage section.
01  wa-length       pic 9(5) comp.
01  wa-in           pic x(50).
01  end-of-file     pic s9(9) comp value external rms$_eof.
procedure division.
declaratives.
dv-fa-in section.
    use after standard exception procedure on fa-in.
end declaratives.
theprogram.
    read fa-in into wa-in
    evaluate true
      when rms-sts of fa-in = end-of-file  s1
      when rms-sts of fa-in < end-of-file  s2
      when rms-sts of fa-in > end-of-file  s2
    end-evaluate
    s3
```

9.4 Test data generation

The solution to a path condition provides a test case that will cause execution of the path. The test case is of one of three forms: file only input; interactive only input; both file and interactive input.

For test cases that are wholly read from files the system generates the necessary files containing the data values produced during feasibility checking. The program is simply executed and no further user intervention is required.

When the test case is wholly input interactively by the user the test cases are placed in file in000.dat. The file contains, in the order of input, the variable name and its data value. At present this file is listed and used as a note pad when executing the program. It is planned to construct a test harness for interactive input which will cause execution without user input so that testing will be undertaken in a uniform manner regardless of the nature of the input mechanism. Testing the quality of the human-machine interface is not under consideration here so the loss of its visibility is not of concern.

For hybrid file and interactive input test cases both the necessary files and in000.dat are created. During execution the user inputs the interactive inputs and the rest are read from the files as usual. Again, the planned test harness will streamline this testing treating interactive input simply as input from another file.

9.5 Intermediate form

The developer of the software under test is not required to examine the transformed source program. It is an intermediate representation used during path selection and symbolic execution. However, both forms can be compiled and executed and will produce equivalent results.

This approach has the advantage of allowing the creation of a comparatively simple translator from source program to intermediate form which is useful for a prototype development allowing easy incremental inclusion of language constructs.

9.6 Summary

The creation of an intermediate form can be achieved either by direct translation from the source program or by several steps of translation from the source into simpler or standard forms in the source language. The final version of the 'source' program is then translated into the intermediate form. The SYM-BOL system uses the second of these two approaches. COBOL has a wide variety of both assignment and branching statements. The core of the translation strategy is the use of three standard forms, one each for arithmetic, input and branching statements.

The first-stage intermediate form is a COBOL program with dramatically reduced statement variety. This form can be compiled and run giving identical results to the original program. This is then translated into the final stage intermediate form similar to that found in other

systems. This two stage approach has the benefit for a prototype system of allowing the development of a stable COBOL-to-intermediate-form translator whilst allowing easy inclusion of an additional COBOL feature by the introduction of a new routine in the first-stage translator which just translates the new COBOL feature into the standard form.

Most of the earlier systems do not undertake the translation into intermediate form in the two staged manner employed by SYM-BOL. Introducing additional language features is thus likely to be a little more difficult in these systems though not a significant problem. The SPADE system is an exception translating the source program into an intermediate form that could be described as a specification language. This form must be further translated into a form suitable for path extraction and symbolic execution. It has the advantage of providing a common intermediate form for many program languages; thus development of a tool for another language is eased at the expense of some additional translation for each language. The approach used in SPADE has the further disadvantage of providing output that refers to the specification language which is further from the source program than is the COBOL standard form from its source program.

CHAPTER TEN PATH SELECTION AND SYMBOLIC EXECUTION

The problem of determining what input data is required to execute a path is in some ways more difficult for COBOL-based systems intended for commercial file processing than for systems based on other languages such as FORTRAN, intended for numerical algorithms. The reason for this is that when a symbolic executor is used in a commercial data processing environment a path requires the reading of records from files. This means that a test case becomes a set of files of records rather than just several numerical values. On the other hand, the path conditions may be simpler than for numerical algorithms which can contain complex predicates.

In a commercial file processing program a single path can give almost complete branch coverage and will cause the processing of many records from several files. There will be many symbolic values generated during symbolic execution of such a path and the ordering of these values is critical to the execution of the path.

Appendix A contains a detailed example of the use of the SYM-BOL system to test a straightforward sequential update program. Only three paths are required to achieve complete branch coverage. Two of these paths cater for the cases of empty files. The bulk of the processing is covered by one path which reads records from a master file and a transaction file and writes records to a new master file and an error listing file. The path that covers the majority of branches requires five records within the master file and five records in the transaction file. The relationship between the records within each of the files is as important as the relationship between (the records in) the two files.

10.1 Path selection and symbolic execution

The term 'path selection' is adopted in favour of the more common 'path generation'. 'Selection' implies a degree of care not suggested by 'generation' which implies an almost random choice.

10.1.1 Path selection version 1

In the early stage of development the only method of path selection in the SYM-BOL system was a simple user selection at each branch point. The user is presented with the alternative branch predicates and makes a selection. The appropriate predicate is placed on the straight-line form of the path followed by the statements on the branch. This is repeated for each branch in turn until an exit is reached.

The straight-line form of the path is now symbolically executed using forward expansion to produce variable expressions and the path condition. The completed path condition is checked for feasibility. Whilst this method worked it became clear that many of the causes of infeasible paths could be avoided if the user had visibility of the expressions for the variables in the predicates being considered. To achieve this visibility symbolic execution must be performed in step with path selection rather than at the culmination of path selection.

10.1.2 Path selection version 2

At each branch point the predicates for each branch are displayed together with the current expression for each variable in the predicates. The user can now see that some of the

predicates are clearly infeasible. For example:

```
Node 006  on  Path 002

1 002  J = I
2 000  J > I
3 001  J < I

Current expression for J
A@01 + 5

Current expression for I
A@01
```

In this case it is clear that branches 1 and 3 are infeasible and branch 2 is feasible. An additional aid is the provision of the branch coverage count, 002, 000, 001 in the example. If all the branches were feasible the user may choose to select branch 2 as it has not yet been covered.

Where the expressions are too complicated for the user to make a quick judgement on the feasibility by inspection a branch may be chosen based on the branch coverage, or even at random. The system then assesses its feasibility. When an infeasible choice is made the expressions are redisplayed and an alternative branch is chosen. This is repeated until a feasible branch is identified.

In some cases the expressions are constants. For example:

```
Node 007  on  Path 002

1 002  X = Z
2 000  X > Z
3 001  X < Z

Current expression for X
5

Current expression for Z
5
```

Here the only feasible branch is branch 1. The system now makes this choice without presenting it to the user. The predicate is not conjoined to the path condition as it is true and therefore redundant.

10.2 Automatic path selection

By utilizing the current expressions for the variables and the branch coverage data an automatic path selector has been created. First, predicates which are obviously infeasible are removed from consideration. Second, the branch coverage value is used to select the least covered of the remaining branches. However, this strategy is flawed. Many of the uncovered parts of a program may require passage through a well exercised branch before they can be reached. For this form of automatic path selection to be successful the forward expansion approach must be partly abandoned and replaced by a strategy that commences at an uncovered branch. Path selection now works forward from this point in the usual way and backward to the beginning of the program. This 'uncovered-branch-out' approach has not been implemented in the SYM-BOL system partly because it requires the addition of a backward substitution symbolic executor and partly because there is another promising alternative for automatic path selection based on domains. However, the uncovered-branch-out approach to branch selection is planned to be added to a later version because there are always some branches which are difficult to place on a feasible path and this strategy will be of value in attempting to place them on a feasible path.

Over the past few years, researchers have been considering the problem of what constitutes

a good test. This work has resulted in the use of domains as a means of creating test cases [Whit85]. Where a test case can be considered to represent a large set of possible test cases then it can be seen as a better test case than one which is representative of only a small number of possible cases. The SYM-BOL system is designed to incorporate a path selection strategy in which the expressions produced by the symbolic execution are utilised in an attempt to identify 'interesting' paths. For example, a path which constrains variables less than another path is potentially more important to the tester because a larger domain has been examined. The basis of the strategy is to maximize the coverage of each variable's domain.

10.2.1 Active and idle domains

Consider the COBOL variable declared as follows:

```
01  A PIC S999.
```

This declaration specifies the domain of the variable. It may take on values from -999 to 999 and the domain of A may be expressed as:

```
Domain of A  -1000 < A < 1000
```

For a particular path the PC is:

```
PC   -10 < A < 25
```

It is clear that the domain of the variable A on the path is constrained to a small subset of the domain of variable A. This may be termed the active domain of variable A. The non-covered parts of the domain, which may be termed the idle domain of variable A, may be represented by the following:

```
-1000 < A < -9
   24 < A < 1000.
```

If further paths exist which would transfer all or part of an idle variable domain to an active

variable domain they should be included in the test set. There is a risk that this strategy will produce a large number of paths. Thus, the strategy of the path selector will need to incorporate a means of choosing branches that minimize the size of idle variable domains.

A path that minimizes the idle domain for one variable is unlikely to have the same effect on all other input variables on the same path. A compromise here may be to determine both the sum of the minimum values of the input variables and the sum of the maximum values of the input variables covered so far at each conditional statement. The branch that provides the greatest range between minimum and maximum sums of input variables is the branch selected for inclusion on the path being generated.

Once complete branch coverage has been achieved the variable ranges would be examined for idleness. Further paths may be selected to reduce the size of idle domains. A final refinement requires that the set of paths is now reduced, where possible, to decrease the number of paths without decreasing coverage nor increasing idleness. The first version of the SYM-BOL system uses only a simple domain coverage strategy when selecting paths and does not attempt to minimise the idle domain sizes once the paths have been created.

10.3 User v automatic path selection

Indications so far suggest that the goal of automatic path selection is difficult to achieve if good selection decisions are to be made. A skilled (and even a not so skilled) software tester will devise better tests than the automatic selection approaches described here and elsewhere.

The difficulty facing the tester when selecting paths and creating test cases is an administrative one. For example, it is difficult to: keep track of expressions for variables and path conditions, determine path feasibility, generate test cases to satisfy the path condition, create the files of test cases and execute interactive programs with the intended inputs. The most useful software testing tools will be the ones that give the greatest support to the tester rather than the ones that are intended to replace the tester. The SYM-BOL system can be used in either capacity but the supporting role looks the more promising.

10.4 Summary

In some of the earlier systems (ATTEST, CASEGEN, FORTRAN testbed) path selection and symbolic execution are undertaken in isolation. First, a path or set of paths is identified. Second, the paths are symbolically executed to produce a path condition. As each branch predicate is conjoined to the path condition the newly created partial path condition is passed to either a theorem prover or an optimizer to assess whether the path is feasible. This approach is useful as it identifies the point at which the path becomes infeasible and allows the selection of an alternative branch without the need for recommencing the path selection from the entry point. But, what it lacks is the ability to avoid selecting the branch that causes infeasibility in the first place. A simple inspection is often sufficient to show that a branch predicate would cause infeasibility.

The SYM-BOL system integrates path selection and symbolic execution allowing the current variable expressions and the branch coverage measures to be used in path selection. This

helps achieve branch coverage and reduces the selection of infeasible paths. Both EFFIGY and IPS also provide output of the path condition and variable expressions allowing the user to avoid selection of obviously infeasible paths.

SYM-BOL provides both user and automatic path selection modes, the integrated path selection and symbolic execution being used in both modes of operation. First indications are that the most useful mode is the user selection as this overcomes the branch selection problem of what may be called 'busy bottlenecks'. Here, a small number of branches must be executed many times to allow all the branches beyond these points to be executed at least once. Busy bottlenecks are easily identified and handled by the user but no method is incorporated within the automatic mode of SYM-BOL, nor any other automatic system, to deal with this situation. Interestingly, neither EFFIGY nor IPS provide automatic path selection.

CHAPTER ELEVEN DETERMINING PATH FEASIBILITY AND TEST DATA GENERATION

The identification of feasible and infeasible paths is central to the path-based approach to testing. There are two approaches to determining infeasibility: axiomatic and algebraic [Clar81].

The axiomatic technique makes use of a theorem proving system to determine whether the constraints are contradictory [Mann73]. The algebraic technique uses the simple conditions within the path condition as a set of constraints which the system attempts to solve. The COBOL symbolic execution testing system uses the algebraic technique.

When using the algebraic approach an artificial objective function is created, for example, the sum of the variables present in the predicates. An optimizer is then used in an attempt to minimise the objective function subject to the constraints. The objective function selected does not appear to affect whether a solution can be found only the nature of the solution. Non-linear sets of constraints for a path are problematic as there is no assurance that a solution can be found even though the constraints are not in conflict. Linear systems of constraints do not pose this problem and output from a linear optimizer will be in one of two forms:

1. A solution, in which case the path is feasible. The solution may be used as a test case to execute the path.
2. A message indicating that the constraints are contradictory the path is infeasible.

White and Sahay [Whit85] coin the phrase 'linearly domained program' to identify the class

of programs amenable to the use of linear optimizers for feasibility checking. White and Cohen [Whit80] have identified Commercial Data Processing software as a class of software for which non-linear constraints are unlikely to pose a problem.

White and Cohen report that in a study of 50 COBOL programs from data processing applications the most important result was that only one predicate out of the 1225 tabulated was non-linear. They go on to conclude: "we believe the sample is large enough to indicate that non-linear predicate interpretations are rarely encountered in data processing applications." This may be because Commercial Data Processing rarely requires squaring, cubing and higher order functions.

In an analysis of COBOL programs [Alja79] the ADD statement accounted for 73.5% of all arithmetic statements. The verb that facilitates squaring and cubing is COMPUTE. In the study COMPUTE accounted for only 3.2% of all arithmetic statements. Unfortunately, no analysis of the use of the COMPUTE verb was given but it seems reasonable to assume that only a small proportion of the arithmetic statements contained squaring, cubing or higher order functions. If this is general then the occurrence of non-linear predicates is likely to be negligible.

White and Cohen concluded: "It is clear that any testing strategy restricted to linear predicates is still viable in many areas of programming practice". Systems, such as symbolic executors, which can determine infeasibility by using linear optimizers will be useful for a large number

of programs and in particular for Commercial Data Processing applications.

11.1 The problem of the alphanumeric literal and the linear optimizer

Formulating the path predicates into a problem suitable for a linear optimizer is a relatively straightforward matter for numerical programs where predicates tend to consist only of operators, numeric variables and numeric constants. Commercial Data Processing software on the other hand contains an additional class of predicate component - the alphanumeric. Constraints that contain only numeric variables, numeric constants and alphanumeric variables but not alphanumeric literals can be processed by a numerical optimizer treating all variables as numeric. The solution will not necessarily be a natural test case but its feasibility will have been assessed. The problematic path condition is the one that contains alphanumeric literals which cannot sensibly be passed to an optimizer which requires numeric data.

Two alternative approaches to overcoming this problem are:

1. For each alphanumeric literal in a predicate create a numeric token. Pass the numeric token, along with the variables and constants, to the optimizer.
2. Separate the predicates of the path condition into two categories: those containing alphanumeric literals and variables and those not containing alphanumerics. Formulate the numeric predicates into an optimization problem and pass to an optimizer. The alphanumeric predicates are now evaluated by a specially written routine.

Both of these approaches rely on the fact that predicates containing alphanumeric variables

are independent of the predicates containing numeric variables. This is clearly true as it is meaningless to compare an alphanumeric variable with a numeric variable. Of course, the value of an alphanumeric variable may at some stage determine the value of a numeric variable such as a condition based on an alphanumeric variable where one branch modifies a numeric variable but this is taken in to account by the symbolic execution of the path.

Consider the following fragment of code:

```
 1      begin-example.
 2          accept ST1
 3          accept ST2
 4          accept A
 5          if A > 15
 6          then
 7            compute C = C + 1
 8          else
 9            if ST1 = "X"
10            then
11              compute B = B + 2
12            else
13              compute B = B + 1
14            end-if
15          end-if
16          if A < 10
17          then
18            compute D = D + 1
19          end-if
20          if ST2 = "Y"
21          then
22            compute C = C + 2
23          end-if
24          if ST1 = ST2
25          then
26            compute D = B + C
27          end-if
28          stop run.
29      end program example.
```

There are many paths through this program fragment. Consider the following path: 1, 2, 3, 4, 5, 8, 9, 10, 11, 14, 15, 16, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28. The statements on this path and its evolving path condition as it is symbolically executed by forward expansion are

as follows:

```
   Statement            Path Condition

 2 accept ST1
 3 accept ST2
 4 accept A
 5 [not A > 15]         not A > 15
 9 [ST1 = "X"]          not A > 15, ST1 = "X"
11 compute B = B+2
16 [not A < 10]         not A > 15, ST1 = "X", not A < 10
20 [ST2 = "Y"]          not a > 15, ST1 = "X", not A < 10, ST2 = "Y"
22 compute C = C + 2
24 [ST1 = ST2]         not A > 15, ST1 = "X", not A < 10, ST2 = "Y", ST1 = ST2
26 compute D = B + C
```

11.1.1 Alternative 1 The numeric token approach

Substitute "X" by 1 and "Y" by 2 giving the new set of constraints:

```
not A > 15, ST1 = 1, not A < 10, ST2 = 2, ST1 = ST2
```

All the path constraints are now submitted to the numerical optimizer where the problem cannot be solved hence the path is infeasible.

Suppose now that the condition on line 24 had read: if ST1 > ST2. The constraints would now be:

```
not A > 15, ST1 = "X", not A < 10, ST2 = "Y" and ST1 > ST2
```

Substitute "X" by 5 and "Y" by 2 giving the new set of constraints:

```
not A > 15, ST1 = 5, not A < 10, ST2 = 2 and ST1 > ST2
```

This is solvable yet the path is infeasible. The introduction of a simple rule overcomes the difficulty: numerical tokens must give the same sort sequence as the literals they represent. Applying this rule would mean that tokens to represent "X" and "Y" must reflect the relationship "X" < "Y", the tokens 5 and 2 violate this principle. Changing the values to say, 4 and 6 respectively, overcomes the problem giving constraints of:

```
not A > 15, ST1 = 4, not A < 10, ST2 = 6 and ST1 > ST2
```

which is infeasible.

11.1.2 Alternative 2 The alphanumeric constraint solver

Separate the path constraints into two classes.

	numeric		alphanumeric
N1	not A > 15	A1	ST1 = "X"
N2	not A < 10	A2	ST2 = "Y"
		A3	ST1 = ST2

The numeric constraints are input to a numerical optimizer and yield a solution. The right hand sides of A1 and A2 are substituted into A3 giving:

```
A4 "X" = "Y"
```

Clearly this expression is contradictory. The path is thus infeasible.

The case above is trivial. The approach is useful only if it is practical for larger sets of constraints. Consider the following set of constraints:

```
1.  AN1 > 'X'
2.  AN2 < 'Y'
3.  AN3 = 'Z'
4.  AN1 > AN2
5.  AN3 = AN4
6.  AN4 = AN2
```

For each equality containing a variable on one side of the equality substitute the expression on the other side into all other constraints containing the variable. Duplicate constraints should not be introduced. Several passes through the constraints should be made until no new constraints can be created. The substitution should terminate as soon as a contradictory constraint is produced.

```
3. in 5. gives  7.   'Z' = AN4
5. in 6.        8.   AN3 = AN2
6. in 2.        9.   AN4 < 'Y'
6. in 4.       10.   AN1 > AN4
3. in 8.       11.   'Z' = AN2
5. in 9.       12.   AN3 < 'Y'
5. in 10.      13.   AN1 > AN3
7. in 9.       14.   'Z' < 'Y'
```

Constraint 14. is contradictory so the path is infeasible.

On completion of repeated substitution from equalities contradictions may not have been detected although they are present. Substitution from inequalities must be undertaken e.g.

```
                1.   AN1 > 'X'
                2.   AN2 < 'X'
                3.   AN1 < AN2
1. in 2. gives  4.   AN2 < AN1
3. in 4.        5.   AN2 < AN2
```

Constraint 5. is contradictory so the path is infeasible.

Many substitutions may be necessary before feasibility is assured. An assessment of the maximum number of substitutions possible shows the potential size of the problem.

The number of variables and literals (operands) determines the maximum number of constraints. For two operands there are four possible constraints: <, >, = and <>. Where constraints consist solely of two operands then the maximum number of constraints possible is given by:

```
Max Con = (N(N-1)/2)*4 = 2N(N-1)   where N is the number of operands.
```

The maximum number of substitutions is given by:

```
Max Sub = M(M-1)/2   where M is the number of constraints.
```

By substituting Max Con into M the maximum number of substitutions is:

```
Max Sub = 2N(N-1)(2N(N-1)-1)/2
        = N(N-1)(2N(N-1)-1)
        = 2N**4 - 4N**3 + N**2 + N where N is the number of operands.
```

Experience suggests that most condition statements containing alphanumeric literals contain the operators = and <>, but rarely, < and >. This reduces the number of substitutions but does not affect the rate of increase.

Where either side of an inequality may contain expressions of more than one operand the maximum number of substitutions rises even faster.

The creation of a strategy to solve this problem faces the same difficulties as the solution of Linear Programming problems with the additional difficulty of not being able to manipulate the constraints in a straightforward fashion.

The construction of a special routine to deal with alphanumeric literals appears to be impractical. Further research into the nature of path constraints would be needed to establish that the number of:

1. operands present in the conditions of most paths is small;
2. substitutions required are small in comparison with the potential number.

Worley [Worl88], in a response to a paper by the author [Cowa88a], provides an almost linear-time algorithm for undertaking the required substitutions. However, he concludes "unfortunately, neither Coward's algorithm nor this one circumvents the major difficulty... the general problem is NP-complete. Thus, there is no good algorithm for handling the general case". Alternative 2 has been rejected.

Alternative 1 using numeric tokens is the approach implemented in the SYM-BOL system and it appears to work in practice.

11.2 Record and Group Items

A further requirement that must be satisfied to enable the use of linear optimizers for assessing path feasibility concerns the comparison of records with different component structures. Consider the following record declarations.

```
01   A.
     02   B    pic XXX.
     02   C    pic XX.

01   D.
     02   E    pic X.
     02   F    pic XX.
     02   G    pic XX.
```

Suppose a path has constraints:

```
A = D, C not = G.
```

When attempting to solve these constraints the solver must be supplied with the relationship between A and C and between D and G. This can be achieved by submitting only elementary items to the solver. In this case the constraints would be:

```
B + C = E + F + G, C not = G.
```

This is still inadequate. Components that are individually compared need to be isolated giving constraints of:

```
B = E + F, C = G, C not = G.
```

This is adequate to identify the contradiction.

11.3 Formulating LP constraints

Each of the WHEN clauses in the evaluate statement consists of a simple condition. Each simple condition is of one of the forms: A = B, A < B, A > B. In optimization problems it is usual to reformat such constraints as follows:

A = B reformat to A - B = 0

The optimizer used by the author for development work requires constraints to be expressed with both lower and upper bounds.

Thus A = B would be expressed as: 0 <= A - B <= 0

Inequalities are expressed in a similar manner.

To express X < 0 as X <= ? a negative number close to zero is used: -1.0E-21.

To express X > 0 as X => ? a positive number close to zero is used: +1.0E-21.

Where there is no lower or upper bound extremely small or extremely large values are used: -1.0E+21, +1.0E+21. The inequalities would be reformatted as follows:

```
A < B  would be:              -1.0E+21 <= A - B <= -1.0E-21
A > B  would be:              +1.0E-21 <= A - B <= +1.0E+21
       or alternatively:      -1.0E+21 <= B - A <= -1.0E-21
```

Variable declarations provide implicit constraints.

Suppose that variable A is declared as follows: A pic 9.

This would be formulated as a general constraint: 0 <= A <= 9

The optimizer used for development work does not make the input of an objective function mandatory. Where it is omitted the first feasible solution is returned, ideal for the purpose of feasibility checking. For a routine that requires an objective function any expression involving all the variables on the path condition will suffice, say the sum of these variables.

11.4 An example

Consider the program in figure 11.1. Applying the transformations described above to this program yields the program in figure 11.2.

Consider a path through the program in figure 11.2 with the following path condition:

```
I@1 + J@1 > K@1,  J@1 + K@1 > I@1,  K@1 + I@1 >J@1, I@1 = J@1,  J@1 = K@1,
K@1 = I@1,
I@2 + J@2 > K@2, J@2 + K@2 > I@2, NOT K@2 + I@2 >J@2.
```

This path reads two records, the string "Equilateral" is displayed after the first, "Not a Triangle" after the second. (Note that `I@1` represents the first value read into `I`, `I@2` the second value etc.) This path condition together with the variable declarations can be expressed as the following LP problem:

```
Identification Division.
Program-id. triangle.
Environment Division.
Input-Output Section.
File-Control.
    select fa-in assign to "tri.dat".
Data Division.
File Section.
fd fa-in record varying depending wa-length.
01  fa-input        pic x(3).
Working-Storage Section.
01  wa-in.
    03  I           pic 9.
    03  J           pic 9.
    03  K           pic 9.
01  wa-length       pic 9 comp value 3.
01  wa-eof          pic x.
01  tri-match       pic 9.
Procedure Division.
begin-triangle.
    open input fa-in
    read fa-in into wa-in
      end
        move "y" to wa-eof
      not end
        move "n" to wa-eof
    end-read
    perform test before until wa-eof = "y"
      if I + J > K and J + K > I and K + I > J
      then
        move 0 to tri-match
        if I = J
        then
          compute tri-match = tri-match + 1
        end-if
        if J = K
        then
          compute tri-match = tri-match + 1
        end-if
        if K = I
        then
          compute tri-match = tri-match + 1
        end-if
        evaluate true
          when tri-match = 0     display "Scalene"
          when tri-match = 1     display "Isosceles"
          when tri-match = 3     display "Equilateral"
          when other             display "Error"
        end-evaluate
      else
        display "Not a Triangle"
      end-if
      read fa-in into wa-in
        end
          move "y" to wa-eof
      end-read
    end-perform
    close fa-in
    stop run.
end program triangle.
```

Figure 11.1 Triangle program.

```
Identification Division.
Program-id. triangle.
Environment Division.
Input-Output Section.
File-Control.
    select fa-in assign to "tri.dat".
Data Division.
File Section.
fd fa-in record varying depending wa-length.
01  fa-input        pic x(3).
Working-Storage Section.
01  wa-in.
    03  I           pic 9.
    03  J           pic 9.
    03  K           pic 9.
01  wa-length       pic 9 comp value 3.
01  wa-eof          pic x.
01  tri-match       pic 9.
01  end-of-file     pic s9(9) comp value external rms$_eof.
Procedure Division.
declaratives.
dv-fa-in section.
    use after standard exception procedure on fa-in.
end declaratives.
themain section.
begin-triangle.
    open input fa-in
    read fa-in into wa-in
    evaluate true
      when rms-sts of fa-in = end-of-file
        move "y" to wa-eof
      when rms-sts of fa-in > end-of-file
        move "n" to wa-eof
      when rms-sts of fa-in < end-of-file
        move "n" to wa-eof
    end-evaluate.
iter1.
    evaluate true
      when wa-eof < "y"
        continue
      when wa-eof = "y"
        go end-iter1
      when wa-eof > "y"
        continue
    end-evaluate
```

Figure 11.2 Transformed triangle program.

```
      evaluate true
        when I + J > K
          evaluate true
            when J + K > I
              evaluate true
                when K + I > J
                  move 0 to tri-match
                  evaluate true
                    when I = J
                     compute tri-match = tri-match + 1
                  end-evaluate
                  evaluate true
                    when J = K
                      compute tri-match = tri-match + 1
                  end-evaluate
                  evaluate true
                    when K = I
                      compute tri-match = tri-match + 1
                  end-evaluate
                  evaluate true
                    when tri-match = 0     display "Scalene"
                    when tri-match = 1     display "Isosceles"
                    when tri-match = 3     display "Equilateral"
                    when tri-match = 2     display "Error"
                    when tri-match > 3     display "Error"
                  end-evaluate
                when K + I <= J
                  display "Not a Triangle"
              end-evaluate
            when J + K <= I
              display "Not a Triangle"
          end-evaluate
        when I + J <= K
          display "Not a Triangle"
      end-evaluate
      read fa-in into wa-in
      evaluate true
        when rms-sts of fa-in = end-of-file
          move "y" to wa-eof
        when rms-sts of fa-in > end-of-file
          continue
        when rms-sts of fa-in < end-of-file
          continue
      end-evaluate
      go iter1.
  end-iter1.
      close fa-in
      stop run.
  end program triangle.
```

Figure 11.2 Transformed triangle program (continued)

```
Objective function          None

General constraints         0 <= I@1 <= 9
                            0 <= J@1 <= 9
                            0 <= K@1 <= 9
                            0 <= I@2 <= 9
                            0 <= J@2 <= 9
                            0 <= K@2 <= 9

Specific constraints        +1.0E-21 <= I@1 + J@1 - K@1 <= +1.0E+21
                            +1.0E-21 <= J@1 + K@1 - I@1 <= +1.0E+21
                            +1.0E-21 <= K@1 + I@1 - J@1 <= +1.0E+21
                            0 <= I@1 - J@1 <= 0
                            0 <= J@1 - K@1 <= 0
                            0 <= K@1 - I@1 <= 0
                            +1.0E-21 <= I@2 + J@2 - K@2 <= +1.0E+21
                            +1.0E-21 <= J@2 + K@2 - I@2 <= +1.0E+21
                            -1.0E+21 <= K@2 + I@2 - J@2 <= 0
```

The initial point which is thought to be feasible can be assumed to be values of zero for each variable.

The output is a feasible solution, for example:

```
I@1 = 1, J@1 = 1, K@1 = 1, I@2 = 0, J@2 = 1, K@2 = 0.
```

Because a feasible point is identified there are no contradictory constraints, the path is therefore feasible. The values that represent the feasible point can be used as a test case which will cause execution of the identified path.

11.5 Summary

COBOL programs rarely, if ever, contain non-linear path conditions so it is appropriate to use a linear programming routine to assess path feasibility. SYM-BOL uses the NAG-library linear optimizer E04MBF. An advantage of this optimizer is that it can be used to return the first feasible solution rather than an optimal solution, thus reducing the time to assess feasibility. The use of optimizers is not in itself new. What is new is their application to COBOL programs which bring two problems not previously described in the literature.

The first problem is the existence of alphanumeric literals (string constants) on the path condition. Linear programming optimizers require numerics not strings. The solution adopted in SYM-BOL requires the substitution of numeric tokens in place of the alphanumeric literals in such a way that the sort sequence of the original strings is reflected in the sequence of the numeric tokens.

The second problem concerns record and group data items. Consider the following record declarations:

```
01 A1.
   03 A2 pic x.
   03 A3 pic x.
01 B1.
   03 B2 pic x.
   03 B3 pic x.
```

A predicate containing the variable A1, implicity references the variables A2 and A3. For example, the predicate A1 = B1 contains the two implied predicates: A2 = B2 and A3 = B3. Where a path condition contains the predicates A1 = B1 and A2 > B2 then the implied predicates are significant. It is only by considering the implied predicates that the path infeasibility, caused by A2 = B2 and A2 > B2, will be identified. A symbolic execution testing system must, therefore, automatically include all implied predicates or, alternatively, process all variables as a series of characters. The second of these two approaches is discussed in Chapter 12.

There is a third problem which concerns acceptable classes of constraints. Linear programming optimizers cannot process constraints of the form A < > B, only constraints of the forms A < = B and A > = B are generally acceptable to such routines. This problem is

not described in the literature. Branches with such constraints are replaced by two branches A < B, down one branch, and A > B down the other. Not only does this solution overcome the problematic branch predicate but it can also be used to force improved boundary testing. For example, consider the following statement where 'a' and 'b' are unsigned numerics: if a = b. This statement should be tested with cases: a = b, a + 1 = b and a - 1 = b.

To achieve good boundary testing the transformation incorporated in the first-stage translation into standard form creates the branches: a = b, a > b and a < b. By minimizing the objective function, a + b, for each branch the following three sets of values for a and b will be produced: 0,0; 0,1 and 1,0. These values represent good boundary tests.

The literature does not discuss the problems of assessing the feasibility of path conditions containing alphanumeric literals, implicit constraints created by group items and records and constraints of the form A < > B. None of the earlier systems cater for these situations. SYM-BOL is thus novel in catering for all of them.

CHAPTER TWELVE REDEFINITION, REFERENCE MODIFICATION AND STRING HANDLING VERBS

COBOL has a facility for redefinition of data structures such that an item of data may be referenced in a variety of ways. At the most simple a variable may be referenced by several identifiers. More complex redefinition repackages variables such that some characters referenced by a first variable are grouped with other characters of a second variable to constitute a third variable. Reference modification allows the referencing of sub-strings within a string variable without the need for previous definition of an identifier to correspond to the substring. The verbs STRING, UNSTRING and INSPECT provide the more sophisticated means within the language for string manipulation.

All of these features have implications for symbolic execution. These features of the language are not processed by the current SYM-BOL system. This chapter proposes a means of including them within the system.

12.1 Redefines and reference modification

Inclusion of these features in the SYM-BOL system requires the maintenance of expressions for individual characters rather than variables. Consider the following record declarations:

```
01  r1.
    02 z              pic x.
    02 row1.
       03 a           pic x(6).
       03 b           pic x(4).
    02 row2 redefines row1.
       03 j           pic x(4).
       03 k           pic x(6).
01  r2.
    02 x              pic x(6).
    02 y              pic x(4).
```

Consider now the following assignment statement:

```
move k to x
```

this is equivalent to:

```
move row1(5:6) to x
```

Consider now another assignment statement:

```
move b to y
```

this is equivalent to:

```
move row2(7:4) to y
```

which is also equivalent to:

```
move k(3:4) to y
```

One means of creating a unified approach to this variety of means of accessing a series of characters is to maintain symbolic expressions for each character. Every character is referenced using the record name and an appropriate offset subscript and string length. For example, `move b to y`

would be transformed to:

```
move r1(8:4) to r2(7:4)
```

which is processed as a series of individual move statements as follows:

```
move r1(8:1)  to r2(7:1)
move r1(9:1)  to r2(8:1)
move r1(10:1) to r2(9:1)
move r1(11:1) to r2(10:1)
```

Where a value is accepted into a variable, for example,

```
accept b
```

the variable is given a symbolic value say:

```
b@01
```

Where this is then assigned by a move statement such as:

```
move b to y
```

the symbolic value is maintained in the destination value character by character as follows:

```
r2(7:1)  = b@01(1:1)
r2(8:1)  = b@01(2:1)
r2(9:1)  = b@01(3:1)
r2(10:1) = b@01(4:1)
```

Consider the following series of statements based on the declarations above:

```
1     accept a
2     accept b
3     move k to x
```

Line 1 causes the following symbolic values to be assigned:

```
r1(2:1)  = a@01(1:1)
r1(3:1)  = a@01(2:1)
r1(4:1)  = a@01(3:1)
r1(5:1)  = a@01(4:1)
r1(6:1)  = a@01(5:1)
r1(7:1)  = a@01(6:1)
```

Line 2 causes the following symbolic values to be assigned:

```
r1(8:1)  = b@01(1:1)
r1(9:1)  = b@01(2:1)
r1(10:1) = b@01(3:1)
r1(11:1) = b@01(4:1)
```

Line 3 causes the following symbolic values to be assigned:

```
r2(1:1) = a@01(5:1)
r2(2:1) = a@01(6:1)
r2(3:1) = b@01(1:1)
r2(4:1) = b@01(2:1)
r2(5:1) = b@01(3:1)
r2(6:1) = b@01(4:1)
```

The virtue of this verbose solution is that it maintains all the necessary detail for symbolic execution of redefined and reference modified variables. Records (variables declared at the 01 level) can also be redefined. This can be handled by treating all record declarations as one large super-record of which each defined record is a component. The current version of the SYM-BOL system determines the appropriate offset within this super-record and is thus partially prepared for the necessary modifications to incorporate character by character referencing of symbolic values.

12.2 String, Unstring and Inspect

Of all the COBOL features these three verbs pose the greatest difficulty for symbolic execution. Consider the following record declarations and procedural fragment.

```
01  source-string     pic x(10).
01  string-record.
    02 delim          pic x.
    02 dest-string    pic x(10).
    02 point          pic 99.

1   move "f" to delim
2   move "abcdefghij" to source-string
3   move 1 to point
4   unstring source-string
5     delimited delim
6     into      dest-string
7     pointer   point
8   end-unstring
```

After execution of this program fragment the variables contain the following values:

```
source-string  "abcdefghij"
delim          "f"
dest-string    "abcde"
point          7
```

Symbolic execution of this program fragment could mimic the execution and produce the same results. Suppose now that line 1 is changed to:

```
accept delim
```

There are 10 possible outcomes dependent on the value input to 'delim'. The values of 'point' and 'dest-string' after execution with the input shown for 'delim' are as follows:

delim	point	dest-string
'	11	" "
a	2	" "
b	3	"a "
c	4	"ab "
d	5	"abc "
e	6	"abcd "
f	7	"abcde "
g	8	"abcdef "
h	9	"abcdefg "
i	10	"abcdefgh "
j	11	"abcdefghi "
k	11	" "

To express this symbolically requires either a new notation or a transformation for the string verb. The latter looks more promising. Consider the following which is a transformed version of the program fragment above:

```
move 1 to point
perform until point = 11 or source-string(point:1) = delim
  add 1 to point
end-perform
evaluate true
  when point = 11
    move space to dest-string
  when point = 1
    move space to dest-string
    add 1 to point
  when point > 1 and point < 11
    move source-string(1:point - 1) to dest-string
    add 1 to point
end-evaluate
```

This will be further transformed to replace 'perform' and 'add' by the standard forms.

A major effect of this transformation of the string verb is that path selection is now charged with the responsibility of deciding on where, if at all, the delimiter is to be found within the source string. Deciding on when to exit from the loop implicitly decides where the delimiting string is to be located. This is a satisfactory solution as a reasonable number of important boundary positions can be selected without the need for exhaustive coverage. This is in keeping with the path-based approach to testing. The above example caters for a single delimiter of only one character but the principle holds for multiple delimiters of size greater than one character.

Similar transformations are required for 'string' and 'inspect'.

12.3 Summary

The issue of string handling in symbolic execution is not discussed in the literature and no existing system appears to cater for string handling. Two facilities are provided in COBOL for string handling: reference modification and the string handling verbs.

Reference modification provides the ability to reference specific ranges of characters regardless of the grouping of characters determined by variable declarations. This runs contrary to the axiomatic principle of symbolic execution where symbolic expressions are maintained for each variable and no other unit of storage such as individual characters. However, there is no reason to suppose that symbolic execution cannot be extended to maintain symbolic expressions for individual characters. The means of achieving this are outlined earlier in the chapter.

The string handling verbs provide powerful string processing functions. One approach is to treat them as I/O boundaries and create new symbolic expressions noting the output expression. The insights and ingenuity of the user may be employed at this point by placing additional constraints on the new symbolic expression - in effect dynamically creating assertions. A second alternative is to transform the string handling verbs into a series of more elementary constructs each of which is more amenable to symbolic execution. A third approach uses the constructs from the second alternative to derive a set of predefined paths through the expanded construct. A PC and a set of variable expressions is produced for each path. Each of these paths is now treated as a possible branch when undertaking the symbolic

execution. All of these approaches are novel ones for symbolically executing string handling functions.

The whole issue of transforming high-level constructs into lower-level forms suitable for symbolic execution raises an interesting question about the role of symbolic execution in higher level languages. Does symbolic execution have a role in testing software written in a 4GL ? This issue is briefly addressed in the concluding remarks of the final chapter (13).

CHAPTER THIRTEEN SUMMARY AND FURTHER WORK

13.1 Research aims

The research aims (reproduced from the front of the thesis) were:

1. Undertake a literature survey to review the existing symbolic execution testing systems and establish their strengths and weaknesses.
2. Identify the problems facing the application of symbolic execution to commercial data processing software in particular to COBOL. This has not previously been undertaken by another researcher.
3. Propose means of overcoming the problems in creating a COBOL symbolic execution testing system.
4. Devise an approach to path selection that:
   a. selects more useful paths than existing symbolic execution systems;
   b. utilises the results of symbolic execution in a bid to reduce the likelihood of selecting infeasible paths. Existing systems do not do this.
5. Identify problems facing the use of a linear programming routine to assess the feasibility of paths and to overcome these problems demonstrating the practicality of the technique in a COBOL system.
6. Demonstrate that these proposals are practicable by constructing a prototype symbolic execution testing system for COBOL. No such system currently exists for COBOL or any other commercial data processing language. It is not considered possible within the time-scale of a Ph.D. to build a full system.
7. Identify further work necessary to turn the prototype into a full working system and to identify areas in need of further research.

Each of these aims is considered in turn.

13.1.1 Existing symbolic execution testing systems

The associated published paper [Cowa88b] provides the only published review of existing symbolic execution systems. No system has previously been built to execute symbolically commercial data processing software. Of the systems that have been built many omit an assertion processing feature and many leave path selection entirely to the user without making use of the results of symbolic execution up to the branch under consideration. This is caused by these systems treating path selection and symbolic execution as independent activities, symbolic execution being undertaken after the whole path has been identified.

13.1.2 Problems of applying symbolic execution

There are several general problems facing its application. These include:

* ambiguous array references;
* path explosion when symbolically executing into called sub-programs;
* loop processing.

There are further problems more specific to the application of symbolic execution to COBOL. These include:

* large size of language, in particular large variety of branching constructs;
* several files of many records constitute a single test case for a path;
* mixture of interactive user input and reading of file data in a single test case;
* presence of strings in predicates to be solved when assessing path feasibility;
* redefinition of record structures and reference modification;
* string processing using string handling verbs.

The SYM-BOL system caters for most of these problems. Exceptions are redefinition, reference modification and the string handling verbs. Proposals for amending the system to cope with these features have been devised.

13.1.3 Large construct variety in COBOL

The large size of the language has been overcome by defining a core set of COBOL features. A set of standard forms has been established, one for each of: assignment; branching; and arithmetic calculations. By constructing a translator as the first part of the SYM-BOL system, a COBOL program using a wide range of COBOL features is transformed into an equivalent COBOL program containing only the standard forms.

The symbolic execution component of SYM-BOL processes only the standard forms. This has two main benefits. First, development of the SYM-BOL system could be carried out without concern for all the features of COBOL yet the system would process a working COBOL program. More COBOL features can be processed by adding translation functions to the first stage translator without the need to change the symbolic executor. Second, future changes to COBOL will require additions and changes only to the first-stage translator unless there are fundamental changes to the language. This is unlikely as COBOL has a long history of slow (perhaps too slow) change and most old features are supported in later language standards. Other systems make a single step translation into intermediate form. In the event of the addition of a new language construct or a modification to an existing language feature the whole translation software will be subject to modification.

13.1.4 Path selection

Both automatic and user path selection mechanisms are included in the system. The automatic selector uses a combination of current variable expressions, branch coverage and variable domain coverage criteria to choose the next branch to be added to the path. The aim of the strategy is to achieve selection of feasible paths, coverage of each branch and the production of paths that cover large variable domains.

Both automatic and user path selectors utilize the benefits of undertaking path selection and symbolic execution together. The result is avoidance of selection of infeasible paths caused by branch predicates being infeasible based upon their current expressions. Immediate feasibility checking of the path condition once a branch has been added also prevents the creation of infeasible paths as an alternative branch can be selected as soon as the path condition becomes infeasible. This co-ordinated approach to path selection and symbolic execution is a significant advance on many of the earlier systems which treated the two processes separately one after the other. As a result, the creation of infeasible paths is reduced.

13.1.5 Using linear programming for feasibility checking

Fortunately, COBOL programs rarely exhibit non-linear predicates; thus linear programming optimizers are appropriate for feasibility checking. Predicates containing string constants are a problem as a string cannot be passed to a numeric optimizer. This has been overcome by replacing string constants by numeric tokens. The substitution has only one rule. The original

sort sequence of all such substituted strings must be maintained. This transformation is sufficient to allow the successful use of a linear optimizer for feasibility checking. Previous research has not identified strings as a problem for feasibility checking using linear optimizers.

13.1.6 Practicality of a COBOL symbolic execution testing system

The SYM-BOL system is not a complete system exhibiting all the desired features. Nevertheless, it is at such a stage of development that it provides all the expected features of symbolic execution such as:

* creates a path condition for a path;
* creates expressions for output variables;
* determines path feasibility;
* creates test data for a path;
* verifies simple assertions.

The system also includes some new features summarized earlier in this chapter:

* symbolically executes COBOL programs;
* assesses feasibility of path conditions containing strings;
* co-ordinates path selection and symbolic execution so reducing the number of infeasible paths generated;
* displays branch coverage at each branch selection;
* maintains details of inter-relationships of records in files.

There are some problems and impractical aspects of a COBOL symbolic execution testing system. The main ones are:

* presence of constructs of a high level of abstraction such as sort, inspect...;
* presence of string handling facilities making the maintenance of variable expressions rather cumbersome.

These are not insurmountable. A more pertinent question is whether the means of overcoming the impracticalities outweigh the benefits to be gained. The maintenance of the relationship between records in different files is at the heart of testing commercial data processing software. SYM-BOL keeps track of these aspects and whilst, in practice, it is not a trivial matter it is not conceptually difficult. The benefits are the documentation of the testing activity which is a valuable benefit not outweighed by the need to maintain the information. The major weaknesses of SYM-BOL are primarily those of symbolic execution in general rather than of symbolic execution applied to COBOL. Symbolic execution is in some ways more useful when applied to COBOL programs because it can keep track of the error prone task of record interrelationships, a feature not required by other classes of software.

13.2 Strengths of the SYM-BOL system

Predicates containing string constants cannot normally be passed to a numeric optimizer as part of a system of constraints representing the path condition. An effective technique has been devised to allow substitution of these string constants with numeric tokens.

Path generation and symbolic execution are undertaken together. This allows the intermediate results from the symbolic execution to help in path selection and reduces the risk of selecting infeasible paths.

During path selection branch coverage is maintained allowing selection of uncovered branches in preference to already covered branches.

Commercial data processing software exhibits multiple input and output files. The system maintains symbolic values for each field in each record in each input file and expressions for each field in each record in each output file.

In addition to generating feasible path conditions the system also devises test data to cause execution of the path. This is organised into data files such that the program under test can simply be executed without the need for user intervention to create the test files.

13.3 Weaknesses of the SYM-BOL system

In its current prototype form SYM-BOL exhibits a number of weaknesses.

First, the number of COBOL constructs that may be used in the source program is restricted. Some of the accepted constructs are accepted only in limited forms. It is however debatable as to whether this is a weakness of the SYM-BOL system or of the COBOL language itself.

Second, validation of the input source program for non-acceptable constructs and the quality of diagnostic messages is weak.

Third, the system does not record the state of the path condition and variables at each branch point. This is necessary when the system is to provide replay and retrace facilities such as those provided by EXDAMS [Balz69]. However, this was not established as a fundamental requirement of the system and it would not be a difficult omission to rectify.

Fourth, the analysis of the intermediate form for path selection and symbolic execution is based on the transformed 'source' program rather than the original source program. Whilst this has some advantages it also means that the user is presented with information that is not necessarily congruent with the source program submitted to the system. This would not be a problem if the information was cross-referenced to the original source program but at present no cross-referencing is undertaken.

Fifth, the user interface is rather crude, driven from a menu in teletype mode, with, as a result, poor screen handling.

Sixth, because the system was constructed as a prototype it suffers from: excessively modified modules in need of rewriting; modules that contain only minor differences from other modules; inelegant algorithms which could be improved significantly.

Finally, perhaps COBOL is not regarded as one of the best languages for building software tools. However, many of the alternatives had major weaknesses such as not easily supporting the implementation of data hiding within modules. COBOL posed no such problems and is quite suitable for much of the housekeeping processing that is required.

13.4 Further work on the SYM-BOL system

At present a large subset of the COBOL language is acceptable as input to the SYM-BOL system. To increase this to the point where all COBOL programs conforming to the 1985 standard can be submitted to the SYM-BOL system a number of enhancements need to be made:

* validation and filtering of excluded features;
* allowing all variations of 'perform';
* simplification of complex expressions;
* expansion of working storage capacity;
* transform all forms of evaluate to the evaluate standard form;
* maintain the state of the PC and variables at each branch point allowing retrace and replay.

These are comparatively minor upgrades to the current system.

More major work includes catering for:

* relative and indexed files;
* record redefinition;
* the string processing verbs and reference modification.

Arguably, the perennial symbolic execution problem of ambiguous array references could be added to the list but this seems unhelpful as there seems to be a consensus opinion among many researchers which regards this requirement for the availability of dynamic execution data values by a static analysis strategy as insurmountable in any sensible way.

Following improvements to the SYM-BOL system research into the use of the system in practice is required. This could be an extensive research project in itself.

First, general assessment of the usefulness of the tool in practice should be undertaken to determine which facilities and information are useful, which redundant and which missing from the system. One could argue that this should have been established in the early stages of the research but as with all problem solving activities there is a need for review and reassessment.

Second, an analysis of the two modes of path selection, automatic and user, should be made. Over the duration of this research the author has gradually changed his view. At the outset he had blind faith in the notion of a fully automated path and test case generator. Now he

accepts that only some of the user's useful heuristics can be sensibly incorporated into an automated testing system. Further, the more useful tools will be the ones that best support the user, in preference to those that attempt to replace the user. This requires further investigation.

Third, an empirical investigation into the number of loop iterations needed to achieve given levels of coverage is required. A hypothesis for testing could be that only one path is required through a typical commercial data processing program to achieve total branch coverage. The level and nature of the deviation from this premise are of interest.

Fourth, undertake an investigation to determine the situations which would benefit and those which would suffer from symbolically executing modules using macro-expansion or creation of an I/O boundary. Identification of situations which would benefit from the application of both techniques would also be of interest.

Fifth, evaluate the practicality of developing a set of standard assertions for inclusion in file processing software.

Sixth, establish standard forms for high-level constructs such as 'sort' and 'unstring', and establish key paths through the standard form for inclusion on the PC.

13.5 Symbolic Execution of the General Features of Programming Languages

The theory of programming languages provides a set of general program features. Figure 13.1 contains an illustrative, though not necessarily complete, list of such features. Not all of these features are to be found in all languages. Nevertheless, when assessing the usefulness of a technique it is helpful to consider it in terms of the general features of programming languages rather than to limit the discussion to one particular language. The final sections of this thesis briefly consider the usefulness of symbolic execution when applied to programs in general and also the effectiveness of the SYM-BOL system for COBOL programs.

Figure 13.1 lists the general features of programming languages and shows whether symbolic execution in general, and the SYM-BOL system in particular, deal effectively with them. The list is split into two parts. The first contains the elementary features involving declaring, initializing and reassigning values to variables plus the basic flow of control mechanisms and simple I/O. The second part contains the more difficult areas of procedural decomposition and data structuring. It is clear that the part one features are effectively handled by symbolic execution .The effectiveness of symbolic execution in dealing with the part two features is not quite so clear cut and is worthy of further discussion. SYM-BOL caters for nearly all of the part one and many of the part two language features that are implemented in COBOL. This provision is superior to most of the other symbolic execution tools.

	Symbolic execution in general	SYM-BOL in particular
PART ONE - ELEMENTARY FEATURES		
names, references and values	Y	Y
synonyms	Y	Y
typing, declarations and initializations	Y	Y
assignment	Y	Y
expressions	Y	Y
lazy evaluation of logical expressions	Y	Y
keyboard, screen and printer I/O	Y	Y
secondary storage I/O, files	Y	Y
flow of control constructs		
sequence	Y	Y
unconditional non-returning transfer of control	Y	Y
selection		
2-way branch	Y	Y
n-way branch	Y	Y
iteration		
fixed number of iterations	Y	N
pre-condition loop	Y	Y
post-condition loop	Y	Y
PART TWO - DECOMPOSITIONAL FEATURES		
module call		
global-only call	Y	Y
parameterless call	Y	N
parameter-global call	Y	N
parameter-only call	Y	Y
function	Y	N/A
procedure	Y	Y
parameter passing		
call by value	Y	Y
call by reference	Y	Y
call by name	D	N/A
recursion	D	N/A
parallelism	D	N/A
data structuring		
strings	D	D
arrays	D	D
files (direct access)	D	N
records	Y	Y
files (sequential)	Y	Y
dynamic structures and pointer references	D	N/A
sets	D	N/A

Y Effectively handled
N Not handled
D Effectiveness not clear cut- discussed in section 13.5
N/A not applicable - COBOL does not support the feature

Figure 13.1 Ability of symbolic execution to cope with the general features of programming languages and provision in SYM-BOL

13.5.1 Module Calls

A module call is any invocation of out-of-line code. At the point where control is passed to the out-of-line code there is a choice over whether to use macro-expansion or the lemma approach. This does not cause a difficulty but merely presents a choice of approaches. By considering the position of the variable declarations for variables used in both the calling program and the called routine a spectrum of module calls can be identified ranging from the global-only to the parameter-only.

The simplest module call may be termed a 'global-only call'. Here, the variables declared in the calling program are the only variables used in the module i.e. only global variables are in use. Symbolic execution in general and also the SYM-BOL system can sensibly deal with these routines, the PC and variable expressions are updated in the usual way.

The next module call can be described as the 'parameterless call'. Here data is passed between called and calling programs by using global variables, but local variables may also be in use within the module. When the same identifier is used for both local and global variables symbolic execution must be careful to modify only expressions for local variables. Other than this, the use of local variables does not cause a complication for symbolic execution. The SYM-BOL system does not cater for this form of module call.

The worst module to execute symbolically is what can be termed the 'parameter-global' call. This type of call passes parameters but also produces side effects in global variables. As a result it suffers the same difficulty as the parameterless call but in addition some results are

returned by a parameter mechanism. Care must be taken to avoid amending both local and parameter expresions for variables with the same identifier in both module and calling program. Symbolic execution can cope with this type of module but the SYM-BOL system has not been designed to cater for such module calls.

The neatest module to execute symbolically is the parameter-only module call. Here, the source data received from and the results returned to the calling program are passed only by parameter passing mechanisms. Symbolic execution of the module can be undertaken independently of or in concert with the calling modules. The SYM-BOL system has been designed to process such module calls.

13.5.1.1 Parameter Passing

When a module is invoked the actual parameters provided in the calling program must be supplied to the formal parameters in the module. Barron [Barr77] identifies three main means of passing parameters:

call by value;

call by reference;

call by name.

Pratt [Prat84] identifies more categories of parameter passing but these do not need any additional techniques above those required by the three mechanisms listed above.

A 'call by value' simply supplies a copy of the value in the actual parameter to the formal

parameter. Symbolic execution processes this call by simply copying the expression for the actual parameter to the expression for the formal parameter. This has been implemented in SYM-BOL.

A 'call by reference' supplies a reference (or address) of the actual parameter to the formal parameter. Any change to the formal parameter is a change to the actual parameter as the actual and formal parameter identifiers are in effect synonyms for the same storage location. Symbolic execution can handle this in two ways.

First, it can treat the call in much the same way as for call by value, i.e. the expression for the actual parameter is copied to the expression for the formal parameter. In addition the evaluation of the actual parameter is pushed onto a call stack. Immediately on return of control to the calling program the actual parameter is popped from the stack and the expression for the formal parameter is copied to the expression of the actual parameter.

Second, the actual and formal parameters can be treated as though they were synonyms. Whenever a change is made to a formal parameter it is the expression for the actual parameter which is modified.

As both of these approaches mirror established means of implementing call by reference and their effect is equivalent there is no reason to choose one approach over the other. SYM-BOL is designed to handle call by reference by the second approach using the same mechanism

that handles simple variable redefinition.

A 'call by name' defers the evaluation of actual parameters until they are used. It is the called module that determines when, if ever, they are evaluated. The effect is as though the actual parameter is substituted in place of every occurence of the formal parameter in the called module before execution of the module commences. A simple example of the effect of call by name compared to call by reference is given by Pratt [Prat84]. Consider the program fragment in figures 13.2a and 13.2b based on Pratt's example.

In figure 3.2a the effect of calling module r by reference is to pass the address of m and c(2) into the formal parameters i and j respectively. When module r is executed the final values for i and j are 3 and 9 respectively and these are held in the actual parameters m and c(2). On return to the main program the values in m and c(m) are displayed giving 3 and 7 on the screen, the latter value being unchanged by the call.

In figure 13.2b the same program as the one in figure 13.2a is executed; the only difference between the programs is the parameter passing mechanism which is changed to call by name. (Note that this is not legal COBOL and is given only for illustrative purposes.) This time when module r is called the name of the actual parameters are passed to the formal parameters; thus variables i and j take the values 'm' and 'c(m)' respectively. These expressions, consisting of single variable names, are evaluated afresh as they occur in the called module during its execution.

```
identification division.
program-id. callbyreference.
data division.
working-storage section.
01 x pic 9.
01 m pic 9.
01 filler.
   03 c occurs 9 pic 9.
procedure division.
para1.
    move 9 to c(1)
    move 8 to c(2)
    move 7 to c(3)
    move 2 to m
    call "r" using reference m        out: m    = ^m         in: m    = 3
                   reference c(m)           c(m) = ^c(2)          c(2) = 9
    display m c(m)                    display 3 7
    stop run.


identification division.
program-id. r.
data division.
linkage section.
01 i pic 9.                          in: i = ^m    = 2    out: i = ^m   = 3
01 j pic 9.                              j = ^c(2) = 8         j = ^c(2)= 9
procedure division using i j.
para2.
    add 1 to i                           i = 3
    add 1 to j                           j = 9
    exit program.
end program r.

end program callbyreference.
```

Figure 13.2a The Effect of Call By Reference

When the instruction 'add 1 to i' is encountered it is treated as though the variable i had been textually substituted by the actual parameter, m, giving a new value of m = 2 + 1 = 3. This behaves in the same way as call by reference. Execution of the next line 'add 1 to j' is treated as though it read 'add 1 to c(m)'. This is now evaluated for the first time so that c(3) = 7 + 1 = 8. This result is quite different from call by reference. On return to the main program the values 3 and 8 will be displayed.

```
identification division.
program-id. callbyname.
data division.
working-storage section.
01 x pic 9.
01 m pic 9.
01 filler.
   03 c occurs 9 pic 9.
procedure division.
para1.
    move 9 to c(1)                          c(1) = 7
    move 8 to c(2)                          c(2) = 8
    move 7 to c(3)                          c(3) = 7
    move 2 to m                             m    = 2
    call "r" using name m            out: m    =  'm'       in: m    = 3
                   name c(m)                c(m) = 'c(m)'        c(3) = 8
    display m c(m)                          display 3 8
    stop run.

identification division.
program-id. r.
data division.
linkage section.
01 i pic 9.                            in: i = 'm'          out: i = 3
01 j pic 9.                                j = 'c(m)'            j = 8
procedure division using i j.
para2.
    add 1 to i                              i = m    = 2 + 1 = 3
    add 1 to j                              j = c(3) = 7 + 1 = 8
    exit program.
end program r.

end program callbyname.
```

Figure 13.2b The Effect of Call By Name

Symbolic execution of call by name is achieved by passing the name of the parameter rather than by passing the expression that represents the current evaluation of the parameter name. During execution of the called module whenever a formal parameter is encountered it is substituted by the parameter name. Thereafter symbolic execution continues as usual.

A further means of parameter passing is 'call by value and pass back', alternatively known as 'call by value/result'. This is implemented in much the same way as 'call by value'. On

invocation the contents of the actual parameters are copied into the formal parameters. On exit the contents of the formal parameters are copied back into the actual parameters. This appears to achieve the same results as 'call by reference' but the two methods are not semantically equivalent. Myers [Myer78] gives an example to demonstrate that the four means of parameter passing discussed above can yield four different results. When implementing a symbolic execution tool care must be taken to ensure that the semantic differences between the various parameter passing mechanisms are maintained.

In short there are three fundamental parameter passing mechanisms illustrated by the following:

call "r" using value c(m) value passed 8

call "r" using reference c(m) value passed c(2)

call "r" using name c(m) value passed c(m)

All of these are easily symbolically executed in the normal way. The value '8' can be instantly included in an expression. The value c(2) is replaced by its current expression which is then incorporated into in an expression. The value 'c(m)' undergoes two substitutions prior to inclusion in an expression. First, substitute 'm' by its current expression. Second, substitute the whole of the resulting expression by its current expression. Now it is ready for final inclusion in an expression. All of these steps are part of general symbolic execution and so none of the three parameter passing mechanisms poses a difficulty for symbolic execution.

13.5.1.2 Recursion

"The only difference between a recursive call and an ordinary call is that the recursive call creates a second activation of the subprogram during the lifetime of the first activation" [Prat84]. Compilers cope with this difference by the use of a central stack to store the activation records. Symbolic execution can cope with recursion in much the same way. When a recursive call is made the current activation containing the expressions for all variables is pushed onto the central stack. Parameter passing is handled in the usual way as described in the previous section. On exit form the recursively called module the central stack is popped and the activation yielded replaces the current activation.

Recursion is not supported by COBOL and is, therefore not a feature of SYM-BOL.

13.5.1.3 Parallelism

There are two approaches to executing symbolically concurrent modules: the interleaving and the isolation approaches [Dill88].

The interleaving approach merely combines the concurrent modules to form a larger sequential module. This suffers from a combinatorial explosion of paths which grows exponentially with the number of modules. In addition there are difficulties in demonstrating that the form of interleaving chosen does not overlook a potential behaviour.

The isolation approach symbolically executes the concurrent modules independently then

attempts to show that the concurrent modules cooperate and do not suffer from deadlock. Entry assertions for a module specify the constraints on global constants i.e. the nature of the environment. Exit assertions specify the relationship between a module's local variables on termination. Global invariants specify the relationship between variables in different modules. Path conditions are generated for each module to demonstrate its local correctness. Path conditions used in demonstrating local correctness together with global invariants on the local paths are combined to produce verification conditions. If the verification conditions are feasible then the global invariants are upheld and cooperation between parallel modules is demonstrated.

Demonstrating the absence of deadlock requires the creation of an assertion that countenances deadlock. Dillon [Dill88] lists three such syntactic features. These can be used to generate a blocking assertion. If a path containing a blocking assertion is not contradictory then it is prey to deadlock.

Concurrent execution of modules is not provided in COBOL and so this feature is not provided in SYM-BOL.

13.5.2 Data Structuring

Discussion so far in Section 13.5 has concentrated on symbolic execution's ability to cater for the features of procedural decomposition provided by programminmg languages. Data structuring also provides problems worthy of consideration. Symbolic execution of atomic

data items is reasonably straightforward but compound data items, which are aggregates of simple items, require more sophisticated handling. Sometimes, the aggregates are manipulated as a whole, on other occasions reference to the individual components is required.

Barron [Barr77] classifies data aggregates into two as follows:

Arrays: are aggregates in which the components are identifed by their position within the aggregate.

Structures: are aggregates in which the components are identifed by name.

This classification appears to omit a category for files and dynamic data structures which make use of pointers. A further category, 'sequence', may be defined.

Sequences: are aggregates in which the components are not identifed but are retrieved simply in the order of storage.

Unfortunately, this still leaves sets unclassified. The fact that a set contains unordered items conflicts with the members of the sequence class where order is important. Sets therefore constitute a class of their own.

Sets: are aggregates in which the components are not identifed and are not retrieved by use of the order of storage.

13.5.2.1 Arrays

Arrays, strings and direct access files can all be placed in Barron's array class.

The well known problem of symbolic execution of arrays is the ambiguous array reference. Here, the index is dependent on one or more input variables. As a result it is not resolved which element is to be the destination for an assignment or which element is to be compared in a condition test.

Strings can be processed in two ways. First, they can be processed as a whole and as such can be symbolically executed provided the techniques such as those described in this thesis are employed to deal with difficulties faced in feasibility checking. Second, strings can be processed as though they are an array of characters. In this situation their symbolic execution is subject to the same difficulties as arrays in general.

Records held within direct access files, like elements within arrays and characters within strings, are identified by their position within the file. The key field may be subjected to a hash function to yield the location within the file of the desired record but this merely constitutes a 'coded' identification by position. Again, as with arrays, if during symbolic execution the expression for the desired key is dependent on input variables then ambiguous references result.

Overall, Barron's array class is problematic for symbolic execution for just the reason that constitutes the rule of membership of the class. Identification by position, cannot easily be handled when the position is dependent upon input variables.

13.5.2.2 Structures

Many languages use the term 'record' to refer to structures. Here component items are identified by name. Symbolic execution has no problem in performing the basic processing of items identified by name. The only difficulty that arises is in feasibility checking.

A record structure comprises several elementary items. Each item may be the subject of a component constraint on a path condition. The whole record may also be the subject of one or more constraints. There is a risk that conflicting constraints may go undetected because the relationship between a whole record and its component items is not apparent to the feasibility checker. This can be overcome by breaking down constaints containing whole records into several constraints on its components. Any contradictions will now be apparent during feasibility checking.

13.5.2.3 Sequences

This classification includes sequential files and dynamic data structures which make use of pointers. The main feature of this class is that items are retrieved simply in the order of storage. The problem for symbolic execution is to maintain information about the sequence of the items.

Sequential files can be handled quite simply. Each time a record is read a position counter is incremented. Each symbolic value introduced as result of this read is tagged with the

position counter. Account must be taken of the position counter tag when generating files of test cases ready for execution in order to achieve the correct ordering of records in the file.

When a file is opened symbolic execution notes the opening and the access mode. Should a read be attempted on an unopened file or on a file opened only for output symbolic execution will detect this. Similarly, symbolic execution will detect attempts to write to unopened files or files opened only for input. The end of file test can also be handled by symbolic execution by, if necessary because the language does not provide for such an item, creating a data item to indicate whether the file is open or closed.

The pointer used in maintaining dynamic data structures is simply an address for the storage location of the item. Any list or sequence that is implemented using pointers can also be implemented using a static mechanism such as an array and replacing the pointer by an integer subscript.

Symbolic execution can make use of this alternative implementation by using it as model for dynamic sequences. A relative pointer (subscript) can be maintained based on the starting position of the sequence. Symbolic execution merely increments the relative position each time progression to the next item takes place. The expression for the pointer is used only to indicate the position in the sequence for the purpose of generation of test data. As the pointer value is inaccessible within the program no information is lost by adopting this approach.

The only difficulty with the relative-position-pointer approach occurs when items are deleted from the sequence. There appears to be two possible approaches. First, the whole set of pointer expressions must be appropriately adjusted each time a deletion takes place. Second, the set of pointer expressions are left intact and the details of the deletion noted. Immediately prior to the generation of test data or feasibility assessment the pointer expressions must be adjusted. Both of these approaches are unweidly for large sequences. However, it is unlikely that symbolic execution would be used in the creation of large sequences for validation and verification purposes so in practice this is unlikely to pose a significant problem.

13.5.2.4 Sets

There are two categories of operation on a set. First, a set can be tested for emptiness or to determine whether an item is a member of the set. Second, the contents of a set can be modified by: insert; delete; union; intersection; and difference operations.

When symbolic execution makes a membership test on a set the two possibilities of true and false can be instantly assessed by searching the list of values (expressions) that constitute the list. If a matching expression is found then the predicate is true and the constraint can be ignored. If no matching expression is found then it either indicates that the search argument is not in the set or that the matching expression is not in a recognizable form or that the match cannot be be resolved because of the presence of symbolic values for input variables. In this case the expression being sought is to be conjoined to the PC. This is problematic because no straightforward constraint(s) can be found to represent such a predicate when the

set may contain anything other than a limited number of values.

When symbolic execution encounters an empty test on a set the truth can be resolved simply so long as the number of items in the set has been maintained. In essence this requires the symbolic executor to simulate the implementation of the set constructs. This is similar to the methods required for handling other high-level constructs.

A list of the symbolic expressions that constitute the items in the set can be maintained by the symbolic executor. There is no difficulty in symbolically executing operations that require only the addition of new items so insert and union can be handled successfully. But, again as with testing for set membership, recognition of the expression that represents an item to be deleted is a problem. Thus delete, intersection and difference cannot be guaranteed to be symbolically executed correctly.

13.6 Concluding Remarks

There are two important questions to be answered at this stage. First, can symbolic execution be usefully applied to programming languages in general ? Second, can symbolic execution be applied to commercial data processing software in particular to COBOL programs.

All the elementary features of programming languages can be handled by symbolic execution. Of the more sophisticated decompositional features many can be handled in a straightforward

manner. The more advanced features of module calling such as call by name, recursion and parallelism can be handled with the introduction of additional supporting mechanisms not developed when the technique was first muted some 15 or so years ago.

The one stumbling block for symbolic execution is the array. However, the adoption of the viewpoint that sees determining a value for an ambiguous array reference as a matter for path selection allows the technique to progress and provide useful results otherwise unobtainable.

Whether symbolic execution can be applied to commercial data processing software and in particular to COBOL programs requires the answer to address the following:

* general problems of symbolic execution that apply to COBOL programs;
* specific problems that COBOL brings to symbolic execution;
* unsurmounted problems and their level of detraction from the usefulness of the technique.

The literature describes two main problems facing the general application of symbolic execution. These are:

* ambiguous array references;
* path selection especially loops and module calls.

Earlier it was suggested that ambiguous array references can be handled as another aspect of path selection. By choosing a particular index value a virtual path is selected. This could be accomplished by inserting an n-way branch, where n is the number of elements in the array,

into the program at the point where an ambiguous array reference occurs. Resolving ambiguous array references in practice then becomes a matter of path selection. This view reduces all the general problems to one of path selection.

The notion of path selection as a problem for symbolic execution has stood for some time and needs challenging. Path selection is a part of the structural approach to testing software. Symbolic execution is a technique that creates expressions for variables and isolates the conditions that must be true to cause execution of a particular path. These results are useful when testing software. The goal is to carefully test the software not merely to undertake symbolic execution for its own sake. Whether symbolic execution is used or not, path selection is fundamental to the structural approach to testing. A major problem of path selection is the avoidance of infeasible paths. Symbolic execution can be used to assess path feasibility. Symbolic execution may more properly be viewed as a technique for aiding path selection rather than the more usual view that path selection is a hindrance to symbolic execution.

The specific problems that COBOL brings to symbolic execution revolve around:

* high level of abstraction of many COBOL constructs;
* low level of data referencing possible in COBOL.

Assignments of the form: `move a to b`

are of a low level of abstraction when compared to the following:

```
inspect a tallying in b for all c
```

Data references may be partitioned into two types. Consider the following data declarations and three data references:

```
        a     pic x(10).
        b     pic x(5)  occurs 10.

1.      a
2.      a(5:3)
3.      b(6)
```

The first reference refers to all of the characters assigned to the identifier 'a'. References two and three refer to a subset of the characters assigned to identifiers 'a' and 'b', the fifth, sixth and seventh characters of 'a' and the sixth element of 'b' respectively.

13.6.1 High level constructs

At first sight there seems to be little to be gained from substituting high level constructs with more detailed equivalent code before symbolic execution. This amounts to macro-expansion of functions that are known to be correct which could alternatively be treated as an I/O boundary. On the other hand, the code that is substituted has a set of paths which, if fully explored, would give rise to a set of test cases which would test in some detail the output possibilities of the high level construct. In the same way that symbolic execution systems should give the user a choice over whether to employ macro expansion or to use an I/O boundary for module calls, the same facility may also be provided for higher level constructs. It is not necessarily the module or the construct that is being assessed rather it is the use of the module or construct within the program that is on trial. Not only is this approach useful for COBOL but it also has application in the testing of higher level languages. The tester may be prompted to make choices about the nature of the data in a particular test case. This can be achieved using construct standard forms and path selection.

13.6.2 Low level data referencing

The notion of the identifier as the lowest level data reference is inadequate to cater for string processing. By maintaining expressions for individual characters symbolic execution can cater for all features of COBOL programs. The use of data usages other than display, such as binary and computational, does not impact upon symbolic execution as the procedural part of the source program is unaffected by this hidden data representation.

13.7 Conclusion

It is clear that symbolic execution can be applied to COBOL programs and used to verify simple assertions and generate files of test cases ready for execution. Further research is needed to establish whether the effort required to use such a COBOL tool might be better spent in using an alternative approach to software testing. The SYM-BOL prototype is a strong basis from which to undertake this research.

REFERENCES

Alle76 Allen F.E. and Cocke J., "A program data flow analysis procedure", Communications of the ACM, 1976, Vol 19, No 3, pp137-147.

Alja79 Al-Jarrah M.M. and Torsun I.S., "An empirical analysis of COBOL programs", Software practice and experience, 1979, Vol 9, pp341-359.

Andr79 Andrews D., "Using executable assertions for testing and fault tolerance", Proceedings of the international conference on fault tolerant computing, Wisconsin 1979.

Asir79 Asirelli P., Degano P., Levi G., Martelli A., Montanari U., Pacini G., Sirovich F. and Turini F., "A flexible environment for program development based on a symbolic interpreter", Proceedings of the fourth international conference on software engineering, Munich, 1979, pp251-263.

Barr77 Barron D.W., "An introduction to the study of programming languages", Cambridge University Press, 1977.

Balz69 Balzer R.M., "EXDAMS Extendable debugging and monitoring system", Spring joint computer conference AFIPS conference proceedings Boston Mass, May 1969, pp567-580.

Basi87 Basili V.R., "Comparing the effectiveness of testing strategies", IEEE Transactions on Software Engineering, 1987, Vol SE-13, No 12, pp1278-1296.

Basu75 Basu S.K. and Misra J., "Proving program loops", IEEE Transactions on Software Engineering, 1975, Vol SE-1, No 1, pp76-86.

Bazz82 Bazzichi F. and Spadafora I., "An automatic generator for compiler testing", IEEE Transactions on Software Engineering, 1982, Vol SE-8, No 4, pp343-353.

Beiz90 Beizer B., "Software testing techniques", 2nd edition, Van Nostrand Reinhold, 1990.

Bice79 Bicevskis J., Borzovs J., Straujums U., Zarins A. and Miller E.F., "SMOTL - a system to construct samples for data processing program debugging", IEEE Transactions on Software Engineering, 1979, Vol SE-5, No 1, pp60-66.

Bich82 Bichevskii Y.Y. and Borzov Y.V., "Development of symbolic testing methods for computer programs, AUT. REMOT. R., 1982, Vol 43, No 8, pp1054-1061.

Boye75 Boyer R.S., Elpas B. and Levit K.N., "SELECT - a formal system for testing and debugging programs by symbolic execution", Proceedings of the international conference on reliable software, April 1975, pp234-244.

Budd78 Budd T.A. and Lipton R.J., "Mutation analysis of decision table programs", Proceedings of the conference on information science and systems, 1978, pp346-349.

Budd80 Budd T.A., Demillo R.A., Lipton R.J. and Sayward F.G., "Theoretical and empirical studies on using program mutation to test the functional correctness of programs", Proceedings of the seventh ACM symposium on the principles of programming languages, 1980, pp220-222.

Budd81 Budd T.A., "Mutation Analysis : Ideas, examples, problems and prospects", in Computer program testing, Ed Chandraeskaran B. and Radicchi S., North-Holland, 1981.

Carr86 Carre B.A., O'Neill I.M., Clutterbuck D.L. and Debney C.W., "SPADE - The Southampton program analysis and development environment", Chapter 9 of Software engineering environments, Ed. Sommerville I., Peter Pereginus/IEE, 1986, pp129-134.

Chan79 Chan J. "Program debugging methodology", M.Phil. Thesis, Leicester Polytechnic, 1979.

Chea79 Cheatham T.E., Holloway G.H. and Townley J.A., "Symbolic evaluation and the analysis of programs", IEEE Transactions on Software Engineering, 1979, Vol SE-5, No 4, pp402-417.

Clar76a Clarke L.A., "A system to generate test data and symbolically execute programs", IEEE Transactions on Software Engineering, 1976, Vol SE-2, No 3, pp215-222.

Clar76b Clarke L.A., "A program testing system", Annual conference of the ACM Houstan Texas, October 20-22, 1976, pp488-491.

Clar81 Clarke L.A. and Richardson D.J., "Symbolic evaluation methods - implementations and applications", in Computer program testing, Ed Chandraeskaran B. and Radicchi S., North-Holland, 1981, pp65-102.

Clar83 Clarke L.A. and Richardson D.J., "The application of error-sensitive testing strategies to debugging", ACM SIGPLAN notices, 1983, Vol 18, No 8, pp45-52.

Clar85 Clarke L.A. and Richardson D.J., "Applications of symbolic evaluation", Journal of Systems and Software, 1985, Vol 5, No 1, pp15-35.

Clut88 Clutterbuck D.L. and Carre B.A., "The verification of low-level code", Software Engineering Journal, 1988, Vol 3, No 3, pp97-111.

COBO85 "American National Standard for Information Systems - Programming Language - COBOL". 1985, ANSI X3.23-1985, VI-61.

Coen90 Coen-Parisi A. and De Paoli F., "SYMBAD: a symbolic executor of sequential Ada programs", Safecomp90, Gatwick Airport, 30 October - 2 November 1990, pp105-111.

Coop76 Cooper D.W., "Adaptive testing", Proceedings of the second international conference on software engineering, 1976, pp223-226.

Cowa88a Coward P D, "Determining path feasibility for commercial programs", ACM SIGPLAN Notices, 1988, Vol 23, No 3, pp93-101.

Cowa88b Coward P.D., "Symbolic execution systems - a review", Software Engineering Journal, 1988, Vol 3, No 6, pp229-239.

Dann82 Dannenberg R.B. and Ernst G.W., "Formal program verification using symbolic execution", IEEE Transactions on Software Engineering, 1982, Vol SE-8, No 1, pp43-52.

Darr78 Darringer J.A. and King J.C., "Applications of symbolic execution to program testing", Computer, April 1978, pp51-60.

Dear83 Dearnley P.A. and Mayhew P.J., "In favour of system prototypes and their integration into the systems development cycle", The Computer Journal, 1983, Vol 26, No 1, pp36-42.

DeMa81 De Marco T., "Structured Analysis and System Specification", Yourden Press, 1981.

Demi79 Demillo R.A., Lipton R.J. and Perlis A.J., "Social processes and proofs of theorems and programs", Communications of the ACM, 1979, Vol 22, No 5, pp271-280.

Dill88 Dillon L.K., "Symbolic execution-based verification of Ada tasking programs", Proceedings of 3rd international IEEE conference on Ada applications and environments, IEEE, NH, May 1988, pp3-13.

Dura81 Duran J.W. and Ntafos S.C., "A report on random testing", Proceedings of the fifth international conference on software engineering, 1981, pp179-183.

Dura84 Duran J.W. and Ntafos S.C., "An evaluation of random testing", IEEE Transactions on Software Engineering, 1984, Vol SE-10, No 4, pp438-444.

Elpa72 Elpas B., Levitt K.N., Waldinger R.J. and Waksman A., "An assessment of techniques for proving program correctness", Computing Surveys, 1972, Vol 4, No 2, pp97-147.

Evls73 Evlsamer 1973 quoted in Myers G.J., "Software Reliability Principles and Practices", Wiley, 1976.

Fisc77 Fischer K.F., "A test case selection method for the validation of software maintenance modification", Proceedings of COMPSAC, 1977, pp421-426.

Floy67 Floyd R.W., "Assigning meaning to programs", Proceedings of Symposia in applied mathematics, 1967, Vol 19, p19-32.

Fosd76 Fosdick L.D. and Osterweil L.J., "Data flow analysis in software reliability", Computing Surveys, 1976, Vol 8, No 3, pp305-330.

Gane79 Gane C. and Sarson T., "Structured Systems Analysis : Tools and techniques", Prentice-Hall, 1979.

Gold82 Goldschlager L. and Lister A., "Computer Science: a modern introduction", Prentice-Hall, 1982.

Good75 Goodenough J.B. and Gehart S.L., "Towards a theory of test data selection", IEEE Transactions on Software Engineering, 1975, Vol 1, No 2, pp156-173.

Hanf70 Handford K.V., "Automatic generation of test cases", IBM System Journal, 1970, Vol 9, No 4, pp242-257.

Hans77 Hansen G.A., "Measuring software reliability", Mini-micro systems, 1977, pp54-57.

Hant76 Hantler S.L. and King J.C., "An introduction to proving the correctness of programs", Computing Surveys, 1976, Vol 18, No 3, pp331-353.

Hart90 Hartmann, J. and Robson, D.J., "Techniques for selective revalidation", IEEE Software, 1990, Vol 7, No 1, pp31-36.

Haye87 Hayes I., Specification Case Studies, Prentice-Hall International, 1987.

Hedl81 Hedley D., "Automatic test data generation and related topics", Ph.D. Thesis, Liverpool University, August 1981.

Hedl85 Hedley D. and Hennell M.A., "The cause and effects of infeasible paths in computer programs", Proceedings of the eighth international conference on software engineering, 1985.

Hekm84 Hekmatpoor S. and Ince D.C., "An evaluation of some black-box testing methods", Technical report No 84/7, Computer discipline Faculty of Mathematics Open University Milton Keynes UK, 1984.

Hekm85 Hekmatpoor S., "The execution of formal specifications",Technical report No 85/2, Computer discipline Faculty of Mathematics Open University Milton Keynes UK, 1985.

Henn76 Hennell M.A., Woodward M.R. and Hedley D., "On program analysis", Information processing letters, 1976, Vol 5, No 5, pp136-140.

Henn83 Hennell, M.A., Hedley, D., and Riddell, I.J.: "The LDRA software testbeds: Their roles and capabilities". Proceedings of the IEEE Software Fair 83 Conference Arlington Virginia, July 1983. (Liverpool Data Research Associates, Merseyside Innovation Centre, 137 Mount Pleasant, Liverpool L3 5TF.)

Howd75 Howden W., "Methodology for the generation of program test data", IEEE Transactions on Computers, 1975, Vol C-24, No 5, pp554-559.

Howd77 Howden W., "Symbolic testing and the DISSECT symbolic evaluation system ", IEEE Transactions on Software Engineering, 1977, Vol SE-3, No 4, pp266-278.

Howd78a Howden W., "DISSECT - A symbolic evaluation and program testing system", IEEE Transactions on software engineering, 1978, Vol SE-4, No 1, pp70-73.

Howd78b Howden W., "An evaluation of the effectiveness of symbolic testing", Software - Practice and Experience, 1978, Vol 8, pp381-397.

Howd81 Howden W.E., "Errors, design properties and functional program tests", in Computer program testing, Ed Chandrasekaran B. and Radicchi S., North-Holland, 1981.

Ince85 Ince D.C., "The automatic generation of test data", The Computer Journal, 1985, Vol 30, No 1, pp63-69.

Jack75 Jackson M., "Principles of Program Design", Academic Press, 1975.

Jone86 Jones C. B., "Systematic Software Development using VDM", Prentice-Hall International, 1986.

Kemm85a Kemmerer R.A. and Eckman, S.T., "UNISEX - A unix-based symbolic executor for Pascal", Software - Practice and Experience, 1985, Vol 15, No 5, pp439-458.

Kemm85b Kemmerer R.A., "Testing formal specifications to detect design errors", IEEE Transactions on Software Engineering, 1985, Vol SE-11, No 1, pp32-43.

King75 King J.C., "A new approach to program testing", 1975 International conference on reliable software, April 1975, pp228-233.

King76 King J.C., "Symbolic execution and program testing", Communications of the ACM, 1976, Vol 19, No 7, pp385-394.

King81 King J.C., "Program reduction using symbolic execution", ACM SIGSOFT software engineering notes, 1981, Vol 6, No 1, pp9-14.

Knut73 Knuth D.E. and Stevenson F.R., "Optimal measurement points for program frequency count", BIT, 1973, Vol 13.

Laka76 Lakatos I., "Proofs and refutations: the logic of mathematical discovery", Cambridge University Press, 1976.

Levi83 Levi G. and Pegna A.M., "Top-down mathematical semantics and symbolic execution", R.A.I.R.O. Informatique Theorique, 1983, Vol 17, No 1, pp55-70.

Loo88 Loo P.S. and Tsai W.K., "Random testing revisited", Information and Software Technology, 1988, Vol 30, No 7, pp402-417.

Mann73 Manna Z., Ness S. and Vuillemin J., "Inductive methods for proving properties of programs", Communications of the ACM, 1973, Vol 16, No 8, pp491-502.

Mill63 Miller J.C and Maloney C.J., "Systematic mistake analysis of digital computer programs", Communications of the ACM, 1963, Vol 6, pp58-63.

Mill74 Miller E.F. and Paige M.R., "Automatic generation of software testcases", Eurocomp conference proceedings, 1974, pp1-12.

Mill84 Miller E.F., "Software quality assurance", Presentation, London, 14-15 May 1984.

Myer78 Myers G.J., "Chapter 14 Programming Languages", Composite/structured design, Van Nostrand Reinhold Company Inc, 1978.

Myer79 Myers G.J., "Chapter 4 Test case design", The art of software testing, John Wiley and sons Inc, 1979.

Norm90 Norman R.W., "Essential COBOL", McGraw Hill, 1990.

Ntaf88 Ntafos S.C., "A comparison of some structural testing strategies", IEEE Transactions on Software Engineering, Vol 14, No 6, pp868-874.

O'Ne88 O'Neill G., "Evaluation of the RTP Pascal-MALPAS IL translator", National Physical Laboratory report DITC 128/88, Sept 1988.

O'Ne89 O'Neill I.M. and Clutterbuck D.L., "Tool support for software proof", IEE Coloquium on "The application of computer aided software engineering tools", 1989, Digest No 24, pp8/1-4.

Oste76 Osterweil L.J. and Fosdick L.D., "Some experience with DAVE - A FORTRAN program analyser", AFIPS conference proceedings, 1976, pp909-915.

Oste83 Osterweil L.J., "TOOLPACK - An experimental software development environment research project", IEEE Transactions on Software Engineering, 1983, Vol SE-9, No 6, pp673-685.

Paig74 Paige M.R. and Benson J.P., "The use of Software probes in testing FORTRAN programs", Computer, July 1974, pp40-47.

Park82 Parkin A., "COBOL for students", 2nd edition, Arnold, 1982.

Payn78 Payne A.J., "A formalized technique for expressing compiler exercisers", ACM SIGPLAN notices, 1978, Vol 13, No 1, pp59-69.

Ploe79 Ploedereder E., "Pragmatic techniques for program analysis and verification", Proceedings of the fourth international conference on software engineering, 1979, pp63-72.

Prat84 Pratt T.W., "Programming languages - Design and implementation", Prentice Hall, 2nd Edition, 1984.

Rama74 Ramamoorthy C.V. and Ho S.F., "FORTRAN automatic code evaluation system", Electron. Res. Lab., University California Berkeley, Rep. M-466, Aug 1974, cited by Howd77.

Rama76 Ramamoorthy C.V., Ho S.F. and Chen W.J., "On the automated generation of program test data", IEEE Transactions on Software Engineering, 1976, Vol SE-2, No 4, pp293-300.

Rich81 Richardson D.J. and Clarke L.A., "A partition analysis method to increase program reliability", Proceedings of the fifth international conference on software engineering, 1981, pp244-253.

Selb87 Selby R.W., Basili V.R. and Baker F.T., "Cleanroom software development: An empirical evaluation", IEEE Transactions on Software Engineering, 1987, Vol SE-13, No 9, pp1027-1037.

Spec84 Spector A. and Gifford D., "Case Study: The space shuttle primary computer system", Communications of the ACM, 1984, Vol 27, No 9, pp874-900.

Tayl80 Taylor R.N. and Osterweil R.N., "Anomaly detection in concurrent software by static data flow analysis", IEEE Transactions on Software Engineering, 1980, Vol SE-6, No 3, pp265-277.

Tayl83 Taylor R.N., "An integrated verification and testing environment", Software-Practice and Experience, 1983, Vol 13, pp697-713.

VanT78 Van Tassel D., "Program style, design, efficiency, debugging and testing", Prentice Hall, 1978.

Voge80 Voges V., Gmeiner L. and Amschler A., "SADAT - An automated testing tool", IEEE Transactions on Software Engineering, 1980, Vol SE-6, No 3, pp286-290.

Webb87 Webb J.T., "MALPAS - an automatic static analysis tool for software validation and verification", 1st International Conference on reliability and robustness of engineering software, Ed Brebbia C.A. and Keramidas G.A., 23-25 Sept 1987, pp67-75.

Wein73 Weinberg G. M., "The Psychology of Computer Programming", Van Nostrand Reinhold, 1971.

Weyu80 Weyuker F.J. and Ostrand T.J., "Theories of program testing and the application of revealing subdomains", IEEE Transactions on Software Engineering, 1980, Vol SE-6, No 3, pp236-246.

Weyu82 Weyuker E.J., "On testing non-testable programs", The Computer Journal, 1982, Vol 25, No 4, pp465-470.

Whit80 White L.J. and Cohen E.I., "A domain strategy for computer program testing", IEEE Transactions on Software Engineering, 1980, Vol SE-6, No 3, pp247-257.

Wood80 Woodward M.R., Hedley D. and Hennell M.A., "Experience with path analysis and testing of programs", IEEE Transactions on Software Engineering, 1980, Vol SE-6, No 6, pp278-285.

Yau87 Yau, S.S. and Kishimoto, Z., "A method for revalidating modified programs in the maintenance phase", IEEE COMPSAC, 1987, pp272-277.

Youn88 Young M. and Taylor R.N., "Combining static concurrency analysis with symbolic execution", IEEE Transactions on Software Engineering, 1988, Vol 14, No 10, pp1499-1511.

APPENDICES

Appendix A Testing a sequential update program using SYM-BOL

The SYM-BOL system displays two menus for selecting options within the system. These are shown below. In the following log of a session using SYM-BOL the menus are not repeated each time they would be displayed but instead they are abbreviated to the menu headings.

```
MAIN MENU
=========

0 - Quit
1 - Pre-process COBOL source
2 - Automatic Path Selection
3 - User Path Selection - Feasibility Every Branch
4 - User Path Selection - Feasibility End of Path
5 - Generate Files of Test Data
6 - Execute program under test
7 - Inspect Results Files
8 - Erase SYM-BOL Generated Files

Enter choice [0] :


INSPECTION OF RESULTS FILES MENU
================================

0 - Quit

1 - se2cond.dat
2 -     tok.dat
3 -       *.cob
4 -   pnod*.dat
5 -   path*.dat
6 -    *pc*.dat
7 -      v*.dat
8 -     lp*.*
9 -     t3*.*
10 -    P5*.*
11 - free format

Enter choice [0] :
```

```
ZEU$ se p5                                                    Invoke SYM-BOL to
                                                              analyze p5.cob
                    SYM-BOL

                    COBOL SYMBOLIC EXECUTION TESTING SYSTEM

                    Program under test : P5.COB

                    MAIN MENU
                    =========
Enter choice [0] : 7                                          Inspect Results Files
                    INSPECTION OF RESULTS FILES MENU
                    ================================
Enter choice [0] : 3                                          *.cob

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

P5.COB;1

Total of 1 files.
Which file do you want to view? [exit] : p5.cob               The source program
                                                              to be analyzed
       Identification Division.
       Program-id. p5.
      * Loosely based on a program taken from Parkin
      * 'COBOL for students' 2nd Ed, 1982

       Environment Division.
       Input-output Section.
       File-control.
           select fa-old-master assign to p5om.
           select fb-trans      assign to p5tr.
           select fc-new-master assign to p5nm.
           select fd-print-file assign to p5pr.

       Data Division.
       File Section.

       fd  fa-old-master record varying depending wa-length.
       01  fa-old-master-rec        pic x(17).

       fd  fb-trans record varying depending wb-length.
       01  fb-trans-rec             pic x(18).

       fd  fc-new-master record varying depending wc-length.
       01  fc-new-master-rec        pic x(17).

       fd  fd-print-file record varying depending wd-length.
       01  fd-print-rec             pic x(132).

       Working-Storage Section.
       01  wa-old-master-rec.
           02  wa-customer-no       pic 9(6).
           02  wa-customer-name     pic x(10).
           02  wa-credit-code       pic x.

       01  wa-length          comp  pic 999      value 17.

       01  wb-trans-rec.
           02  wb-trans-type        pic x.
```

```
           88  insertion          value "i".
           88  amendment          value "a".
           88  deletion           value "d".
       02  wb-trans-detail.
           03  wb-customer-no     pic 9(6).
           03  wb-customer-name   pic x(10).
           03  wb-credit-code     pic x.
01     wb-length          comp    pic 999       value 18.
01     wc-length          comp    pic 999       value 17.
01     wd-length          comp    pic 999       value 132.

01     wg-error-messages.
       02  filler                 pic x(36)
           value "insertion - record already on master".
       02  filler                 pic x(36)
           value "no master record for this customer".

01     wg-messages redefines wg-error-messages.
       02  wg-message                  pic x(36)
                                       occurs 2.
01     wg                            pic 99.

01     wh-global-conditions.
       02  wh-eof-old-mast-flag       pic x.
           88  not-eof-old-master     value "f".
           88  eof-old-master         value "t".
       02  wh-trans-eof-flag          pic x.
           88  not-eof-trans          value "f".
           88  eof-trans              value "t".

01     wi-page-heading.
       02  wi-page-heading-1.
           03  filler             pic x(13)   value spaces.
           03  filler             pic x(42)
               value "customer master file update - error report".
           03  filler             pic x(9)    value "     page".
           03  wi-page-counter    pic z9.

       02  wi-page-heading-2.
           03  filler             pic x(31)
               value "cust #    cust name  cred  tr".

01     wj-error-line.
       02  wj-customer-no         pic 9(6).
       02  filler                 pic x(4)  value spaces.
       02  wj-customer-name       pic x(10).
       02  filler                 pic x(4)  value spaces.
       02  wj-credit-code         pic x.
       02  filler                 pic x(4)  value spaces.
       02  wj-trans-type          pic x.
       02  filler                 pic x(4)  value spaces.
       02  wj-message             pic x(36).

01     wk-report-footing.
       02  filler                 pic x(13)    value "end of report".

01     wl-page.
       02  wl-page-counter        pic 99       value 1.
       02  wl-line-counter        pic 99       value 0.
```

```
 Procedure Division.
 a1-update-customer-master.

*    get first transaction record
     open input fb-trans
     move "f" to wh-trans-eof-flag
     read fb-trans into wb-trans-rec
     end
       display "program error a1 - empty transaction file"
       display "customer master file update terminating"
       close fb-trans
       go stop-run
     end-read

*    get first master record
     open input fa-old-master
     move "f" to wh-eof-old-mast-flag
     perform b3-read-fa-old-master

*    initialize error report
     open output fd-print-file
     move w1-page-counter to wi-page-counter
     write fd-print-rec from wi-page-heading-1 after advancing 1
     write fd-print-rec from wi-page-heading-2 after advancing 2
     compute w1-line-counter = w1-line-counter + 3

*    open new master
     open output fc-new-master
     perform b1-process-trans until eof-trans
     close fb-trans
     perform b2-write-and-read-master until eof-old-master
     close fa-old-master fc-new-master
     write fd-print-rec from wk-report-footing after advancing 2
     close fd-print-file
     go stop-run.

 b1-process-trans.
     perform b2-write-and-read-master
       until eof-old-master or wa-customer-no not < wb-customer-no

*      process transaction record
       if not-eof-old-master and wa-customer-no = wb-customer-no
       then
         if amendment
         then

*          update old master
           if wb-credit-code not = space
           then
             move wb-credit-code to wa-credit-code
           end-if
           if wb-customer-name not = space
           then
             move wb-customer-name to wa-customer-name
           end-if
         else
           if deletion
           then
             perform b3-read-fa-old-master
           else
             move 1 to wg
```

```
                perform c1-generate-error-message
              end-if
            end-if
          else
            if insertion
            then
              write fc-new-master-rec from wb-trans-detail
            else
              move 2 to wg
              perform c1-generate-error-message
            end-if
          end-if
          read fb-trans into wb-trans-rec
          end
            move "t" to wh-trans-eof-flag
          end-read.

      b2-write-and-read-master.
            write fc-new-master-rec from wa-old-master-rec
            perform b3-read-fa-old-master.

      b3-read-fa-old-master.
          read fa-old-master into wa-old-master-rec
          end
            move "t" to wh-eof-old-mast-flag
          end-read.


      c1-generate-error-message.
          if w1-line-counter > 20
          then
            compute w1-page-counter = w1-page-counter + 1
            move     w1-page-counter to wi-page-counter
            write fd-print-rec from wi-page-heading-1
                  after advancing page
            write fd-print-rec from wi-page-heading-2
                  after advancing 2
            move 0 to w1-line-counter
          end-if
          move wb-customer-no   to wj-customer-no
          move wb-customer-name to wj-customer-name
          move wb-credit-code   to wj-credit-code
          move wb-trans-type    to wj-trans-type
          move wg-message (wg)  to wj-message
          write fd-print-rec  from wj-error-line after advancing 2
          compute w1-line-counter = w1-line-counter + 2.

      stop-run.
          stop run.
      end program p5.

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

P5.COB;1

Total of  1 files.
Which file do you want to view? [exit] :                    exit
```

```
                    MAIN MENU
                    =========
Enter choice [0] : 1                                    Pre-process COBOL
                                                        source
                    Pre-processing of COBOL source

                    0 - Quit
                    1 - Include Assertions
                    2 - Exclude Assertions

Enter choice [0] : 2                                    Exclude Assertions

Pre-processing of COBOL source  Started
Pre-processing of COBOL source  O.K.

                    MAIN MENU
                    =========
Enter choice [0] : 7                                    Inspect Results Files

                    INSPECTION OF RESULTS FILES MENU
                    ================================
Enter choice [0] : 1                                    se2cond.dat

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

SE2COND.DAT;1

Total of 1 file.
Which file do you want to view? [exit] : se2cond.dat    condition names
                                                        and full form
insertion               wb-trans-type           = "i"
amendment               wb-trans-type           = "a"
deletion                wb-trans-type           = "d"
not-eof-old-master      wh-eof-old-mast-flag    = "f"
eof-old-master          wh-eof-old-mast-flag    = "t"
not-eof-trans           wh-trans-eof-flag       = "f"
eof-trans               wh-trans-eof-flag       = "t"

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

SE2COND.DAT;1

Total of 1 file.
Which file do you want to view? [exit] :                exit

                    INSPECTION OF RESULTS FILES MENU
                    ================================
Enter choice [0] : 2                                    tok.dat

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

TOK.DAT;1

Total of 1 file.
Which file do you want to view? [exit] : tok.dat
```

```
010     page                                                    strings and their
020a                                                            numeric tokens
030cust #    cust name  cred  tr
040customer master file update - error report
050customer master file update terminating
060d
070end of report
080f
090i
100insertion - record already on master
110no master record for this customer
120program error a1 - empty transaction file
130t

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

TOK.DAT;1

Total of 1 file.
Which file do you want to view? [exit] :                        exit

                    INSPECTION OF RESULTS FILES MENU
                    ================================
Enter choice [0] : 3                                            *.cob
Directory $1$DIB1:[LCS.PD_COWARD.CSE]

P5.COB;1            RDIN.COB;1          RDOUT.COB;1          SE1IN.COB;1
SE1OUT.COB;1        SE3OUT.COB;1

Total of 6 files.
Which file do you want to view? [exit] : se3out.cob             The pre-processed
                                                                program
       Identification Division.
       Program-id. p5.
     * Loosely based on a program taken from Parkin
     * 'COBOL for students' 2nd Ed, 1982
       Environment Division.
       Input-output Section.
       File-control.
           select fa-old-master assign to p5om.
           select fb-trans      assign to p5tr.
           select fc-new-master assign to p5nm.
           select fd-print-file assign to p5pr.

       Data Division.
       File Section.
       fd  fa-old-master record varying depending wa-length.
       01  fa-old-master-rec        pic x(17).

       fd  fb-trans record varying depending wb-length.
       01  fb-trans-rec             pic x(18).

       fd  fc-new-master record varying depending wc-length.
       01  fc-new-master-rec        pic x(17).

       fd  fd-print-file record varying depending wd-length.
       01  fd-print-rec             pic x(132).
       Working-Storage Section.
       01  end-of-file comp pic s9(9)  value external rms$_eof.
       01  wa-old-master-rec.
```

```
    02  wa-customer-no        pic 9(6).
    02  wa-customer-name      pic x(10).
    02  wa-credit-code        pic x.
01  wa-length          comp   pic 999      value 17.

01  wb-trans-rec.
    02  wb-trans-type         pic x.
        88  insertion         value "i".
        88  amendment         value "a".
        88  deletion          value "d".
    02  wb-trans-detail.
        03  wb-customer-no    pic 9(6).
        03  wb-customer-name  pic x(10).
        03  wb-credit-code    pic x.
01  wb-length          comp   pic 999      value 18.
01  wc-length          comp   pic 999      value 17.
01  wd-length          comp   pic 999      value 132.

01  wg-error-messages.
    02  filler                pic x(36)
    value "insertion - record already on master".
    02  filler                pic x(36)
    value "no master record for this customer".

01  wg-messages redefines wg-error-messages.
    02  wg-message                 pic x(36)
                                   occurs 2.
01  wg                             pic 99.

01  wh-global-conditions.
    02  wh-eof-old-mast-flag       pic x.
        88  not-eof-old-master     value "f".
        88  eof-old-master         value "t".
    02  wh-trans-eof-flag          pic x.
        88  not-eof-trans          value "f".
        88  eof-trans              value "t".

01  wi-page-heading.
    02  wi-page-heading-1.
        03  filler            pic x(13)   value spaces.
        03  filler            pic x(42)
        value "customer master file update - error report".
        03  filler            pic x(9)    value "     page".
        03  wi-page-counter   pic z9.
    02  wi-page-heading-2.
        03  filler            pic x(31)
        value "cust #    cust name  cred  tr".

01  wj-error-line.
    02  wj-customer-no        pic 9(6).
    02  filler                pic x(4)  value spaces.
    02  wj-customer-name      pic x(10).
    02  filler                pic x(4)  value spaces.
    02  wj-credit-code        pic x.
    02  filler                pic x(4)  value spaces.
    02  wj-trans-type         pic x.
    02  filler                pic x(4)  value spaces.
    02  wj-message            pic x(36).

01  wk-report-footing.
    02  filler                pic x(13)    value "end of report".
```

```
 01  wl-page.
     02  wl-page-counter      pic 99       value 1.
     02  wl-line-counter      pic 99       value 0.


 Procedure Division.
 declaratives.
 ds-fb-trans                  section.
     use after standard exception procedure on
     fb-trans                 .
     ds-fa-old-master             section.
     use after standard exception procedure on
     fa-old-master            .
 end declaratives.
 main section.

 a1-update-customer-master.

*    get first transaction record
     open input fb-trans
     move "f" to wh-trans-eof-flag

     read fb-trans                    into wb-trans-rec
     evaluate true
       when rms-sts of fb-trans                    = end-of-file
        display "program error a1 - empty transaction file"
        display "customer master file update terminating"
        close fb-trans
        go stop-run

       when rms-sts of fb-trans                    > end-of-file
         continue
       when rms-sts of fb-trans                    < end-of-file
         continue
     end-evaluate

*    get first master record
     open input fa-old-master
     move "f" to wh-eof-old-mast-flag
     perform b3-read-fa-old-master

*    initialize error report
     open output fd-print-file
     move wl-page-counter to wi-page-counter
     write fd-print-rec from wi-page-heading-1 after advancing 1
     write fd-print-rec from wi-page-heading-2 after advancing 2
     compute wl-line-counter = wl-line-counter + 3

*    open new master
     open output fc-new-master
     perform loop-1
     close fb-trans
     perform loop-2
     close fa-old-master fc-new-master
     write fd-print-rec from wk-report-footing after advancing 2
     close fd-print-file.
     go stop-run .
 b1-process-trans.
     perform loop-3
*    process transaction record
```

```
evaluate true
when
wh-eof-old-mast-flag        = "f"

evaluate true
when
wa-customer-no              = wb-customer-no

evaluate true
when
wb-trans-type               = "a"


evaluate true
when
wb-credit-code              = space
continue
when
wb-credit-code              > space
move wb-credit-code to wa-credit-code
when
wb-credit-code              < space
move wb-credit-code to wa-credit-code
end-evaluate

evaluate true
when
wb-customer-name            = space
continue
when
wb-customer-name            > space
move wb-customer-name to wa-customer-name
when
wb-customer-name            < space
move wb-customer-name to wa-customer-name
end-evaluate
when
wb-trans-type               > "a"

evaluate true
when
wb-trans-type               = "d"
perform b3-read-fa-old-master
when
wb-trans-type               > "d"
move 1 to wg
perform c1-generate-error-message
when
wb-trans-type               < "d"
move 1 to wg
perform c1-generate-error-message
end-evaluate
when
wb-trans-type               < "a"

evaluate true
when
wb-trans-type               = "d"
perform b3-read-fa-old-master
```

```
when
wb-trans-type              > "d"
move 1 to wg
perform c1-generate-error-message
when
wb-trans-type              < "d"
move 1 to wg
perform c1-generate-error-message
end-evaluate
end-evaluate
when
wa-customer-no             > wb-customer-no

evaluate true
when
wb-trans-type              = "i"
write fc-new-master-rec from wb-trans-detail
when
wb-trans-type              > "i"
move 2 to wg
perform c1-generate-error-message
when
wb-trans-type              < "i"
move 2 to wg
perform c1-generate-error-message
end-evaluate
when
wa-customer-no             < wb-customer-no

evaluate true
when
wb-trans-type              = "i"
write fc-new-master-rec from wb-trans-detail
when
wb-trans-type              > "i"
move 2 to wg
perform c1-generate-error-message
when
wb-trans-type              < "i"
move 2 to wg
perform c1-generate-error-message
end-evaluate
end-evaluate
when
wh-eof-old-mast-flag       > "f"

evaluate true
when
wb-trans-type              = "i"
write fc-new-master-rec from wb-trans-detail
when
wb-trans-type              > "i"
move 2 to wg
perform c1-generate-error-message
when
wb-trans-type              < "i"
move 2 to wg
perform c1-generate-error-message
end-evaluate
when
wh-eof-old-mast-flag       < "f"
```

```
    evaluate true
    when
    wb-trans-type              = "i"
    write fc-new-master-rec from wb-trans-detail
    when
    wb-trans-type              > "i"
    move 2 to wg
    perform c1-generate-error-message
    when
    wb-trans-type              < "i"
    move 2 to wg
    perform c1-generate-error-message
    end-evaluate
    end-evaluate

    read fb-trans                    into wb-trans-rec
    evaluate true
      when rms-sts of fb-trans                    = end-of-file

        move "t" to wh-trans-eof-flag

      when rms-sts of fb-trans                    > end-of-file
        continue
      when rms-sts of fb-trans                    < end-of-file
        continue
    end-evaluate.

b2-write-and-read-master.
    write fc-new-master-rec from wa-old-master-rec.
    perform b3-read-fa-old-master.

b3-read-fa-old-master.

    read fa-old-master               into wa-old-master-rec
    evaluate true
      when rms-sts of fa-old-master               = end-of-file

        move "t" to wh-eof-old-mast-flag

      when rms-sts of fa-old-master               > end-of-file
        continue
      when rms-sts of fa-old-master               < end-of-file
        continue
    end-evaluate.

 c1-generate-error-message.

     evaluate true
     when
     wl-line-counter           = 20
     continue
     when
     wl-line-counter           > 20
     compute wl-page-counter = wl-page-counter + 1
     move wl-page-counter to wi-page-counter
     write fd-print-rec from wi-page-heading-1
     after advancing page
     write fd-print-rec from wi-page-heading-2
     after advancing 2
     move 0 to wl-line-counter
```

```
      when
      wl-line-counter          < 20
      continue
      end-evaluate
        move wb-customer-no to wj-customer-no
        move wb-customer-name to wj-customer-name
        move wb-credit-code to wj-credit-code
        move wb-trans-type to wj-trans-type
        move wg-message (wg) to wj-message
        write fd-print-rec from wj-error-line after advancing 2
        compute wl-line-counter = wl-line-counter + 2.

stop-run.
    stop run.

loop-1.

    evaluate true
    when
    wh-trans-eof-flag        = "t"
    continue
    when
    wh-trans-eof-flag        > "t"
    perform b1-process-trans
    go loop-1
    when
    wh-trans-eof-flag        < "t"
    perform b1-process-trans
    go loop-1
    end-evaluate
      .

loop-2.

    evaluate true
    when
    wh-eof-old-mast-flag     = "t"
    continue
    when
    wh-eof-old-mast-flag     > "t"
    perform b2-write-and-read-master
    go loop-2
    when
    wh-eof-old-mast-flag     < "t"
    perform b2-write-and-read-master
    go loop-2
    end-evaluate
      .

loop-3.

    evaluate true
    when
    wh-eof-old-mast-flag     = "t"
    continue
    when
    wh-eof-old-mast-flag     > "t"

    evaluate true
    when
    wa-customer-no           = wb-customer-no
```

```
            continue
            when
            wa-customer-no              > wb-customer-no
            continue
            when
            wa-customer-no              < wb-customer-no
            perform b2-write-and-read-master
            go loop-3
            end-evaluate
            when
            wh-eof-old-mast-flag        < "t"

            evaluate true
            when
            wa-customer-no              = wb-customer-no
            continue
            when
            wa-customer-no              > wb-customer-no
            continue
            when
            wa-customer-no              < wb-customer-no
            perform b2-write-and-read-master
            go loop-3
            end-evaluate
            end-evaluate
              .

       end program p5.

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

P5.COB;1              RDIN.COB;1          RDOUT.COB;1          SE1IN.COB;1
SE1OUT.COB;1          SE3OUT.COB;1

Total of 6 files.
Which file do you want to view? [exit] :                              exit

            INSPECTION OF RESULTS FILES MENU
            ================================
Enter choice [0] :                                                    exit
                    MAIN MENU
                    =========
Enter choice [0] : 3                                                  User Path Selection
                                                                      - Feasibility every
User Path Selection (Feasibility Every Branch) Started                branch
CLEAR  previous paths [C]
RETAIN previous paths [R]
Enter choice [R] : c
************************************************************
```

```
Node number 001 on Path number001
1 000     fb-trans
        = end-of-file
2 000     fb-trans
        > end-of-file
3 000     fb-trans
        < end-of-file
Current expression for fb-trans
fb-trans@01
Current expression for end-of-file
rms$_eof
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues

End of path 001
Writing              pc001.dat
Writing              path001.dat
Writing              pnod001.dat
Writing              va001.dat

Select another path Y or N ?  [Y] : y
CLEAR  previous paths [C]
RETAIN previous paths [R]
Enter choice [R] : r
************************************************************
Node number 001 on Path number002
1 001     fb-trans
        = end-of-file
2 000     fb-trans
        > end-of-file
3 000     fb-trans
        < end-of-file
Current expression for fb-trans
fb-trans@01
Current expression for end-of-file
rms$_eof
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 002 on Path number002
1 000     fa-old-master
        = end-of-file
2 000     fa-old-master
        > end-of-file
3 000     fa-old-master
        < end-of-file
Current expression for fa-old-master
fa-old-master@01
Current expression for end-of-file
rms$_eof
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
```

```
Node number 003 on Path number002
1 000      wh-trans-eof-flag
         = "t"
2 000      wh-trans-eof-flag
         > "t"
3 000      wh-trans-eof-flag
         < "t"
Current expression for wh-trans-eof-flag
"f"
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 004 on Path number002
Branch 1 is only feasible branch no user choice required
************************************************************
Node number 005 on Path number002
Branch 2 is only feasible branch no user choice required
************************************************************
Node number 006 on Path number002
1 000      wb-trans-type
         = "i"
2 000      wb-trans-type
         > "i"
3 000      wb-trans-type
         < "i"
Current expression for wb-trans-type
wb-trans-type@01
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 007 on Path number002
1 000      fb-trans
         = end-of-file
2 000      fb-trans
         > end-of-file
3 000      fb-trans
         < end-of-file
Current expression for fb-trans
fb-trans@02
Current expression for end-of-file
rms$_eof
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 008 on Path number002
Branch 1 is only feasible branch no user choice required
************************************************************
Node number 009 on Path number002
Branch 1 is only feasible branch no user choice required
************************************************************
End of path 002
Writing              pc002.dat
Writing              path002.dat
Writing              pnod002.dat
Writing              va002.dat
and output file      fd-print-rec002     .dat
and output file      fc-new-master-rec002.dat
```

Throughout the rest of the log nodes with 'no user choice' are reduced to two lines

```
Select another path Y or N ?  [Y] : y
CLEAR  previous paths [C]
RETAIN previous paths [R]
Enter choice [R] : r
**********************************************************
Node number 001 on Path number003
1 001      fb-trans
         = end-of-file
2 000      fb-trans
         > end-of-file
3 001      fb-trans
         < end-of-file
Current expression for fb-trans
fb-trans@01
Current expression for end-of-file
rms$_eof
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
**********************************************************
Node number 002 on Path number003
1 001      fa-old-master
         = end-of-file
2 000      fa-old-master
         > end-of-file
3 000      fa-old-master
         < end-of-file
Current expression for fa-old-master
fa-old-master@01
Current expression for end-of-file
rms$_eof
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
**********************************************************
Node number 003 on Path number003
Branch 3 is only feasible branch no user choice required
**********************************************************
Node number 004 on Path number003
Branch 3 is only feasible branch no user choice required
**********************************************************
Node number 005 on Path number003
1 000      wa-customer-no
         = wb-customer-no
2 000      wa-customer-no
         > wb-customer-no
3 000      wa-customer-no
         < wb-customer-no
Current expression for wa-customer-no
wa-customer-no@01
Current expression for wb-customer-no
wb-customer-no@01
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
**********************************************************
```

```
Node number 006 on Path number003
1 001     fa-old-master
        = end-of-file
2 000     fa-old-master
        > end-of-file
3 001     fa-old-master
        < end-of-file
Current expression for fa-old-master
fa-old-master@02
Current expression for end-of-file
rms$_eof
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 007 on Path number003
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 008 on Path number003
1 000     wa-customer-no
        = wb-customer-no
2 000     wa-customer-no
        > wb-customer-no
3 001     wa-customer-no
        < wb-customer-no
Current expression for wa-customer-no
wa-customer-no@02
Current expression for wb-customer-no
wb-customer-no@01
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 009 on Path number003
Branch 1 is only feasible branch no user choice required
************************************************************
Node number 010 on Path number003
1 000     wa-customer-no
        = wb-customer-no
2 000     wa-customer-no
        > wb-customer-no
3 000     wa-customer-no
        < wb-customer-no
Current expression for wa-customer-no
wa-customer-no@02
Current expression for wb-customer-no
wb-customer-no@01
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is INfeasible -removing last predicate conjoined to PC
************************************************************
```

```
Node number 010 on Path number003
1 000      wa-customer-no
         = wb-customer-no
2 000      wa-customer-no
         > wb-customer-no
3 000      wa-customer-no
         < wb-customer-no
Current expression for wa-customer-no
wa-customer-no@02
Current expression for wb-customer-no
wb-customer-no@01
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 011 on Path number003
1 000      wb-trans-type
         = "a"
2 000      wb-trans-type
         > "a"
3 000      wb-trans-type
         < "a"
Current expression for wb-trans-type
wb-trans-type@01
Select a branch number : 2

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 012 on Path number003
1 000      wb-trans-type
         = "d"
2 000      wb-trans-type
         > "d"
3 000      wb-trans-type
         < "d"
Current expression for wb-trans-type
wb-trans-type@01
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 013 on Path number003
1 001      fa-old-master
         = end-of-file
2 000      fa-old-master
         > end-of-file
3 002      fa-old-master
         < end-of-file
Current expression for fa-old-master
fa-old-master@03
Current expression for end-of-file
rms$_eof
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
```

When system detects infeasible path user is forced to select an alternative branch at the node before progression.

```
Node number 014 on Path number003
1 001      fb-trans
         = end-of-file
2 000      fb-trans
         > end-of-file
3 000      fb-trans
         < end-of-file
Current expression for fb-trans
fb-trans@02
Current expression for end-of-file
rms$_eof
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 015 on Path number003
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 016 on Path number003
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 017 on Path number003
1 001      wa-customer-no
         = wb-customer-no
2 000      wa-customer-no
         > wb-customer-no
3 001      wa-customer-no
         < wb-customer-no
Current expression for wa-customer-no
wa-customer-no@03
Current expression for wb-customer-no
wb-customer-no@02
Select a branch number : 2

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 018 on Path number003
Branch 1 is only feasible branch no user choice required
************************************************************
Node number 019 on Path number003
1 001      wa-customer-no
         = wb-customer-no
2 000      wa-customer-no
         > wb-customer-no
3 000      wa-customer-no
         < wb-customer-no
Current expression for wa-customer-no
wa-customer-no@03
Current expression for wb-customer-no
wb-customer-no@02
Select a branch number : 2

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
```

```
Node number 020 on Path number003
1 000      wb-trans-type
         = "i"
2 000      wb-trans-type
         > "i"
3 000      wb-trans-type
         < "i"
Current expression for wb-trans-type
wb-trans-type@02
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 021 on Path number003
1 000      wl-line-counter
         = 20
2 000      wl-line-counter
         > 20
3 000      wl-line-counter
         < 20
Current expression for wl-line-counter
3
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 022 on Path number003
1 001      fb-trans
         = end-of-file
2 000      fb-trans
         > end-of-file
3 001      fb-trans
         < end-of-file
Current expression for fb-trans
fb-trans@03
Current expression for end-of-file
rms$_eof
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 023 on Path number003
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 024 on Path number003
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 025 on Path number003
1 001      wa-customer-no
         = wb-customer-no
2 001      wa-customer-no
         > wb-customer-no
3 001      wa-customer-no
         < wb-customer-no
Current expression for wa-customer-no
wa-customer-no@03
Current expression for wb-customer-no
wb-customer-no@03
Select a branch number : 1

Checking feasibility of Partial Path
```

```
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 026 on Path number003
Branch 1 is only feasible branch no user choice required
************************************************************
Node number 027 on Path number003
1 001      wa-customer-no
         = wb-customer-no
2 001      wa-customer-no
         > wb-customer-no
3 000      wa-customer-no
         < wb-customer-no
Current expression for wa-customer-no
wa-customer-no@03
Current expression for wb-customer-no
wb-customer-no@03
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 028 on Path number003
1 000      wb-trans-type
         = "a"
2 001      wb-trans-type
         > "a"
3 000      wb-trans-type
         < "a"
Current expression for wb-trans-type
wb-trans-type@03
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 029 on Path number003
1 000      wb-credit-code
         = space
2 000      wb-credit-code
         > space
3 000      wb-credit-code
         < space
Current expression for wb-credit-code
wb-credit-code@03
Select a branch number : 2

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 030 on Path number003
1 000      wb-customer-name
         = space
2 000      wb-customer-name
         > space
3 000      wb-customer-name
         < space
Current expression for wb-customer-name
wb-customer-name@03
Select a branch number : 2

Checking feasibility of Partial Path
```

```
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 031 on Path number003
1 001      fb-trans
         = end-of-file
2 000      fb-trans
         > end-of-file
3 002      fb-trans
         < end-of-file
Current expression for fb-trans
fb-trans@04
Current expression for end-of-file
rms$_eof
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 032 on Path number003
Branch 1 is only feasible branch no user choice required
************************************************************
Node number 033 on Path number003
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 034 on Path number003
1 001      fa-old-master
         = end-of-file
2 000      fa-old-master
         > end-of-file
3 003      fa-old-master
         < end-of-file
Current expression for fa-old-master
fa-old-master@04
Current expression for end-of-file
rms$_eof
Select a branch number : 3

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 035 on Path number003
Branch 3 is only feasible branch no user choice required
************************************************************
Node number 036 on Path number003
1 001      fa-old-master
         = end-of-file
2 000      fa-old-master
         > end-of-file
3 004      fa-old-master
         < end-of-file
Current expression for fa-old-master
fa-old-master@05
Current expression for end-of-file
rms$_eof
Select a branch number : 1

Checking feasibility of Partial Path
Partial Path is Feasible - Symbolic Execution continues
************************************************************
Node number 037 on Path number003
Branch 1 is only feasible branch no user choice required
```

```
End of path 003
Writing              pc003.dat
Writing              path003.dat
Writing              pnod003.dat
Writing              va003.dat
and output file      fd-print-rec003     .dat
and output file      fc-new-master-rec003.dat

Select another path Y or N ?  [Y] : n

User Path (Feasibility Every Branch)  O.K.
                     MAIN MENU
                     =========
Enter choice [0] : 7                                    Inspect Results Files
                     INSPECTION OF RESULTS FILES MENU
                     ================================
Enter choice [0] : 6                                    *pc*.dat

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

PC001.DAT;1          PC002.DAT;3          PC003.DAT;23          SPC001.DAT;1
SPC002.DAT;3         SPC003.DAT;23        TPC001.DAT;1          TPC002.DAT;3
TPC003.DAT;23

Total of 81 files.
Which file do you want to view? [exit] : spc003.dat     path condition for path
                                                         003 with numeric token
022
fb-trans@01 - rms$_eof                  <                    0
fa-old-master@01 - rms$_eof             <                    0
wa-customer-no@01 - wb-customer-no@01   <                    0
fa-old-master@02 - rms$_eof             <                    0
wa-customer-no@02 - wb-customer-no@01   =                    0
wa-customer-no@02 - wb-customer-no@01   =                    0
wb-trans-type@01                        >                  020    "a"
wb-trans-type@01                        =                  060    "d"
fa-old-master@03 - rms$_eof             <                    0
fb-trans@02 - rms$_eof                  <                    0
wa-customer-no@03 - wb-customer-no@02   >                    0
wa-customer-no@03 - wb-customer-no@02   >                    0
wb-trans-type@02                        <                  090    "i"
fb-trans@03 - rms$_eof                  <                    0
wa-customer-no@03 - wb-customer-no@03   =                    0
wa-customer-no@03 - wb-customer-no@03   =                    0
wb-trans-type@03                        =                  020    "a"
wb-credit-code@03 - space               >                    0
wb-customer-name@03 - space             >                    0
fb-trans@04 - rms$_eof                  =                    0
fa-old-master@04 - rms$_eof             <                    0
fa-old-master@05 - rms$_eof             =                    0

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

PC001.DAT;1          PC002.DAT;3          PC003.DAT;23          SPC001.DAT;1
SPC002.DAT;3         SPC003.DAT;23        TPC001.DAT;1          TPC002.DAT;3
TPC003.DAT;23

Total of 81 files.
Which file do you want to view? [exit] :                exit
```

```
                    INSPECTION OF RESULTS FILES MENU
                    ================================
Enter choice [0] : 7                                              v*.dat

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

VA000.DAT;1          VA001.DAT;1          VA002.DAT;1          VA003.DAT;1
VV001.DAT;1          VV002.DAT;1          VV003.DAT;1

Total of 7 files.
Which file do you want to view? [exit] : va003.dat          Variable table
                      In Occ P Siz Val
062
01fa-old-master       05 001 f 000
01fa-old-master-rec   00 001 x 017
01fb-trans            04 001 f 000
01fb-trans-rec        00 001 x 018
01fc-new-master       00 001 f 000
01fc-new-master-rec   00 001 x 017
01fd-print-file       00 001 f 000
01fd-print-rec        00 001 x 132
01end-of-file         00 001 s 009 rms$_eof
01wa-old-master-rec   05 001 x 017
02wa-customer-no      05 001 9 006
02wa-customer-name    05 001 x 010 wb-customer-name@03
02wa-credit-code      05 001 x 001 wb-credit-code@03
01wa-length           00 001 9 003 17
01wb-trans-rec        04 001 x 018
02wb-trans-type       04 001 x 001
88insertion           00 001   000 "i"
88amendment           00 001   000 "a"
88deletion            00 001   000 "d"
02wb-trans-detail     04 001 x 017
03wb-customer-no      04 001 9 006
03wb-customer-name    04 001 x 010
03wb-credit-code      04 001 x 001
01wb-length           00 001 9 003 18
01wc-length           00 001 9 003 17
01wd-length           00 001 9 003 132
01wg-error-messages   00 001 x 072
02filler              00 001 x 036 "insertion - record already on master"
02filler              00 001 x 036 "no master record for this customer"
01wg-messages         00 001 x 072
02wg-message          00 002 x 036
01wg                  00 001 9 002 2
01wh-global-condition 00 001 x 002
02wh-eof-old-mast-flag00 001 x 001 "t"
88not-eof-old-master  00 001   000 "f"
88eof-old-master      00 001   000 "t"
02wh-trans-eof-flag   00 001 x 001 "t"
88not-eof-trans       00 001   000 "f"
88eof-trans           00 001   000 "t"
01wi-page-heading     00 001 x 097
02wi-page-heading-1   00 001 x 066
03filler              00 001 x 013 spaces
03filler              00 001 x 042 "customer master file update - error rep
03filler              00 001 x 009 "     page"
03wi-page-counter     00 001 z 002 1
02wi-page-heading-2   00 001 x 031
```

Variable table after symbolic execution of path 003.
In: number of inputs on path
Occ: num of elements in array
P: picture type
Siz: number of characters
Val: symbolic value at end of path 003

```
03filler                00 001 x 031 "cust #    cust name  cred  tr"
01wj-error-line         00 001 x 070
02wj-customer-no        00 001 9 006 wb-customer-no@02
02filler                00 001 x 004 spaces
02wj-customer-name      00 001 x 010 wb-customer-name@02
02filler                00 001 x 004 spaces
02wj-credit-code        00 001 x 001 wb-credit-code@02
02filler                00 001 x 004 spaces
02wj-trans-type         00 001 x 001 wb-trans-type@02
02filler                00 001 x 004 spaces
02wj-message            00 001 x 036 wg-message[2]
01wk-report-footing     00 001 x 013
02filler                00 001 x 013 "end of report"
01wl-page               00 001 x 004
02wl-page-counter       00 001 9 002
02wl-line-counter       00 001 9 002

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

VA000.DAT;1          VA001.DAT;1          VA002.DAT;1          VA003.DAT;1
VV001.DAT;1          VV002.DAT;1          VV003.DAT;1

Total of 7 files.
Which file do you want to view? [exit] : vv003.dat          Input Variable
                                                            Values
fb-trans@01              0.93E+01   9.3    not eof
rms$_eof                 0.12E+02  12      eof
fa-old-master@01         0.93E+01   9.3    not eof
wa-customer-no@01        0.93E+01   9.3    000009
wb-customer-no@01        0.10E+02  10      000010
fa-old-master@02         0.93E+01   9.3    not eof
wa-customer-no@02        0.10E+02  10      000010
wb-trans-type@01         0.60E+02  60      "d"
fa-old-master@03         0.93E+01   9.3    not eof
fb-trans@02              0.93E+01   9.3    not eof
wa-customer-no@03        0.11E+02  11      000011
wb-customer-no@02        0.87E+01   8.7    000008
wb-trans-type@02         0.10E+02  10      " "
fb-trans@03              0.93E+01   9.3    not eof
wb-customer-no@03        0.11E+02  11      000011
wb-trans-type@03         0.20E+02  20      "a"
wb-credit-code@03        0.11E+02  11      " "
space                    0.87E+01   8.7    space
wb-customer-name@03      0.11E+02  11      "     zzzzz"
fb-trans@04              0.12E+02  12      eof
fa-old-master@04         0.93E+01   9.3    not eof
fa-old-master@05         0.12E+02  12      eof

Directory $1$DIB1:[LCS.PD_COWARD.CSE]

VA000.DAT;1          VA001.DAT;1          VA002.DAT;1          VA003.DAT;1
VV001.DAT;1          VV002.DAT;1          VV003.DAT;1

Total of 7 files.
Which file do you want to view? [exit] :                    exit
                    INSPECTION OF RESULTS FILES MENU
                    ================================
Enter choice [0] :                                          exit
                    MAIN MENU
                    =========
```

```
Enter choice [0] : 5                                    Generate files of
                                                        test data
Test File Generation Started
There are Paths 001 to 003
Enter path number [exit] : 003
Writing       p5om.dat                                  fa-old-master
Writing       p5tr.dat                                  fb-trans
File generation complete for path 003
There are Paths 001 to 003
Enter path number [exit] :                              exit
Test File Generation O.K.
                    MAIN MENU
                    =========
Enter choice [0] : 7                                    Inspect Results Files

                    INSPECTION OF RESULTS FILES MENU
                    ================================
Enter choice [0] : 11                                   Free Format
Free Format - no directory listing
Which file do you want to view? [exit] : p5om.dat       fa-old-master003

000009aaaaaaaaab
000010ccccccccccd
000011eeeeeeeeeef
000012ggggggggggh

Free Format - no directory listing
Which file do you want to view? [exit] : p5tr.dat       fb-trans003

d000010iiiiiiiiiij                                      System has generated
000008kkkkkkkkkkl                                       a file with records
a000011     zzzzz                                       out of sequence.

Free Format - no directory listing
Which file do you want to view? [exit] :                exit
                    MAIN MENU
                    =========
Enter choice [0] : 6                                    Execute Program
Compiling P5.cob                                        Under Test
Linking   P5.obj
Running   P5.exe
                    MAIN MENU
                    =========
Enter choice [0] : 11                                   Free Format
Free Format - no directory listing
Which file do you want to view? [exit] : p5nm.dat       fc-new-master003

000009aaaaaaaaab
000011     zzzzzf
000012ggggggggggh

Free Format - no directory listing
Which file do you want to view? [exit] : p5pr.dat       fd-print-file003
```

```
            customer master file update - error report     page 1
cust #    cust name  cred  tr
000008    kkkkkkkkkk    1         no master record for this customer
end of report

Free Format - no directory listing
Which file do you want to view? [exit] :                    exit
                    MAIN MENU
                    =========
Enter choice [0] :                                          exit

COBOL SYMBOLIC EXECUTION TESTING SYSTEM TERMINATED

ZEU$
```

APPENDIX B SYM-BOL-COBOL: COBOL SUBSET VALID FOR INPUT TO SYM-BOL

Allowed COBOL	Currently Disallowed COBOL
DATA DIVISION	
Sequential files	Indexed files
record description entries redefines occurs	linage clause redefines
level-88 condition-names	
PROCEDURE DIVISION	
Procedure Division	
Procedure Division using	
paragraphs	sections (likely to be marked for deletion in next standard)
Input-output	
accept, display	
open, close	
read, write	start, rewrite, delete
	merge release return
	sort
call	
Assignment	
compute	compute size error add subtract multiply divide
move	set

Flow of control

```
evaluate true
 when
 when
   :
end-evaluate
```

```
evaluate true also true ...
 when
 when
   :
end-evaluate
```

```
evaluate condition
  when true
  when false
end-evaluate
```

```
if simple-condition
then
else
end-if
```

```
if compound-condition
then
else
end-if
```

```
condition-name
```

```
class condition
  e.g. var numeric
```

```
nested if statements
```

```
continue
```

```
go
```

```
perform procedure
```

```
perform procedure varying
```

```
perform until simple-condition
  statement
  :
end-perform
```

```
perform n times
```

```
nested perform statements
```

```
exit program
```

```
stop run
```

Sophisticated string handling

```
string, unstring
inspect
search
```

```
reference modification
  e.g. var(start:length)
```